Writing a Helios Server

Perihelion Software Technical Report No. 8

N. Garnett

April 1989

Perihelion Software Limited
The Maltings
Charlton Road
Shepton Mallet
Somerset
BA4 5QE
England
Telephone +44 749 4203
Fax. +44 749 4977

Contents

1 Introduction

This technical note describes how to write a server for Helios using the Server Library. It largely takes the form of an annotated listing of the current Helios ram disc server which demonstrates most of the features. This note should be read in conjunction with the description of the server library in *The Helios Operating System*. You should also look at the contents of `servlib.h` where the data structures and prototypes of the functions are defined.

2 The Server Library

The Server Library is one of the resident libraries in the Helios nucleus. It provides a number of procedures and management routines to help in the production of Helios servers.

Primarily the server library provides support for a server which needs to maintain a directory structure in memory, it is of little use to servers which maintain off-line directory structures such as filing systems. The directory tree is composed of `DirNode` and `ObjNode` structures which implement directory and terminal nodes respectively. Either may be made an initial substructure of a larger server defined structure. These two structures are substantially similar and may often be used interchangeably in many server library routines.

The server library also provides a message decoder and dispatcher. This waits for messages on a specified port, validates them as GSP messages, and forks a worker process to execute a service procedure. This procedure may either be supplied by the server, or may be one of several standard procedures supplied by the server library itself.

While the server library is a resident part of the nucleus, its definitions are not part of the default set in `/helios/lib/cstart.o`. To access the library you must either add `/helios/lib/servlib.def` during program linkage, or use `/helios/lib/sstart.o` in place of `cstart.o`. Note, however, that `sstart.o` does not include either the C library or Posix library in the link so procedures from these libraries cannot be used. The program's `main` is also entered immediately so no arguments can be passed. The advantage of this is that such servers will run on nodes with only the nucleus present without causing extra code to be loaded.

3 The Ram Disc Server

We start with the header files and some definitions:

```
#include <helios.h>  /* standard header */
#include <string.h>  /* string operations */
#include <codes.h>   /* function & error codes */
#include <syslib.h>  /* system library */
#include <servlib.h> /* server library */
#include <task.h>    /* task structure */

typedef struct File
{
   ObjNode ObjNode;  /* node for directory struct */
} File;

#define Upb     ObjNode.Size     /* use ObjNode size field */
#define Buffers ObjNode.Contents /* use ObjNode contents field */
#define Users   ObjNode.Account  /* number of opens */

#define FileMax 1024  /* bytes in each block */

DirNode Root;         /* root of directory system */

/* Forward procedure definitions */
static File *CreateNode(MCB *m, DirNode *d, string pathname);
void FileRead(MCB *m, File *f);
void FileWrite(MCB *m, File *f);
void FileSeek(MCB *m, File *f);
```

The File structure here contains just the `ObjNode` but in other servers it may contain other fields. The following `#defines` re-define fields of the `ObjNode` structure for convenience. The `ObjNode` field, Contents, is an otherwise unused three word field which in this case is used as a list of data buffers. Root is a static `DirNode` structure which is used as the root of the ram filing system.

```
static void do_open(ServInfo *);
static void do_create(ServInfo *);
static void do_delete(ServInfo *);

static DispatchInfo RamInfo = {
   &Root,       /* Dir tree system root */
   NullPort,    /* request port */
   SS_RamDisk,  /* subsystem code */
   NULL,        /* parent name */
   { NULL, 0 }, /* escape function */
```

```
    {
       { do_open,   2000 }, /* FG_Open */
       { do_create, 2000 }, /* FG_Create */
       { DoLocate,  2000 }, /* FG_Locate */
       { DoObjInfo, 2000 }, /* FG_ObjectInfo */
       { NullFn,    2000 }, /* FG ServerInfo */
       { do_delete, 2000 }, /* FG_Delete */
       { DoRename,  2000 }, /* FG_Rename */
       { DoLink,    2000 }, /* FG_Link */
       { DoProtect, 2000 }, /* FG_Protect */
       { DoSetDate, 2000 }, /* FG_SetDate */
       { DoRefine,  2000 }, /* FG_Refine */
       { NullFn,    2000 }  /* FG_CloseObj */
    }
}
```

This `DispatchInfo` structure is passed to the server library dispatcher. The request port is filled in below, in main, and the parent name is `NULL` (defaults to the current processor name). The main table contains a list of function/stacksize pairs. If an incoming message is one of the standard GSP functions the appropriate entry in the table is forked off with the given stack size. If the message code is outside the standard range, and the escape function is defined, this is called instead, allowing servers to implement private protocols.

```
int main()
{
    char mcname[100];

    MachineName(mcname);

    RamInfo.ParentName = mcname;

    InitNode( (ObjNode *)&Root, "ram", Type_Directory,
       0, DefDirMatrix);

    InitList( &Root.Entries );
    Root.Nentries = 0;

    RamInfo.RegPort = NewPort();
```

The first thing to do in main is to initialise the directory structure by calling `InitNode` on the root directory. It is important that this node has the same name as the server's name in the name table.

The following piece of code does not actually appear in the real ram server. This is because the ram server is usually loaded on demand from disc the

first time a name entry installed by the nucleus is touched. This is not, however, normally the case for other servers which must install their own name in the name server in the following way:

```
    {
       NameInfo Info;
       Object *o = Locate(NULL,mcname);

       Info.Port     = RamInfo.ReqPort;
       Info.Flags    = Flags_StripMame;
       Info.Matrix   = DefDirMatrix;
       Info.LoadData = NULL;

       nte = Create( o, Root->Name, Type_Name,
                     sizeof(NameInfo), (byte *)&Info );

       Close(o);
    }
```

The result of `Create` is an `Object` structure. The name may be removed by `Delete(nte,NULL);`, or simply by calling `Exit` which will do it automatically. Next, we install a parent for the root node and enter the dispatcher.

```
    {
       Object *o;
       LinkNode *Parent;

       o = Locate(NULL,mcname);

       Parent = (LinkNode *)Malloc(sizeof(LinkNode) + strlen(mcname));
       InitNode( &Parent->ObjNode, "..", Type_Link, 0, DefDirMatrix );
       Parent->Cap = o->Access;
       strcpy(Parent->Link,mcname);
       Root.Parent = (DirNode *)Parent;

       Close(o);
    }

    Dispatch(&RamInfo);
    Exit(0);
 }
```

To allow `".."` to work in the root directory the root node's parent is set up as a symbolic link to the current processor. Finally the server library dispatcher is entered. It will only return if the main request port is freed, or other errors start to occur. The call to `Exit` will clean up the server

and cause it to quit. However most servers, once they are running, do not terminate.

The following three procedures are responsible for creating new nodes in the directory tree.

```
File *NewFile(DirNode *dir, string name, word flags, Matrix matrix)
{
   File *f = New(File);

   if( f == NULL ) return NULL;

   InitNode(&f->ObjNode,name,Type_File,flags,matrix);

   InitList(&f->Buffers);
   f->Upb = 0;
   f->Users = 0;

   Insert( dir, &f->ObjNode, TRUE );

   return f;
}

DirNode *NewDir(DirNode *dir, string name, word flags, Matrix matrix)
{
   DirNode *d = New(DirNode);

   if( d == NULL ) return NULL;

   InitNode((ObjNode *)d,name,Type_Directory,flags,matrix);

   InitList(&d->Entries);
   d->Nentries = 0;
   d->Parent = dir;

   Insert(dir, (ObjNode *)d, TRUE );

   return d;
}

static File *CreateNode(MCB *m, DirNode *d, string pathname)
{
   File *f;
   char *name;

   IOCCreate *req = (IOCCreate *)(m->Control);
   name = objname(pathname);

   if( req->Type == Type_Directory )
```

```
      f = (File *)NewDir(d, name, 0, DefDirMatrix );
   else
      f = NewFile(d, name, 0, DefFileMatrix );

   if( f == NULL ) m->MsgHdr.FnRc |= EC_Error | EG_Create;

   return f;
}
```

`NewFile` and `NewDir` are substantially the same, they allocate and initialise an appropriate structure and call Insert to insert the new node in the directory. The third argument to `Insert` indicates whether the directory being added to has been locked by this process. This is the case in both places where `CreateNode` is called below.

The following set of procedures are forked off as processes from the dispatcher. The first is the `Create` operation.

```
static void do_create(ServInfo *servinfo)
{
   MCB *m = servinfo->m;
   MsgBuf *r;
   DirNode *d;
   File *f;
   IOCCreate *req = (IOCCreate *)(m->Control);
   char *pathname a servinfo->Pathname;

   d = GetTargetDir(servinfo); /* find target's parent dir */

   if( d == NULL )
   {
      ErrorMsg(m,EO_Directory);
      return;
   }

   f = (File *)GetTargetObj(servinfo); /* look for target */

   m->MsgHdr.FnRc = SS_Ram; /* re-init return code */

   if( f != NULL ) /* if it exists, moan */
   {
      ErrorMsg(m,EC_Error+EG_Create+EO_File);
      return;
   }

   /* check that we can write to directory, i.e. create a file */
   unless( CheckMask(req->Common.Access.Access,AccMask_W) )
   {
```

```
        ErrorMsg(m,EC_Error+EG_Protected+EO_Directory);
        return;
    }

    r = New(MsgBuf);

    if( r == NULL )
    {
        ErrorMsg(m,EC_Error+EG_NoMemory+EO_Message);
        return;
    }

    f = CreateNode( m, d, pathname ); /* create file/directory */

    if( f == NULL )
    {
        ErrorMsg(m,EC_Error+EG_NoMemory+EO_File);
        Free(r);
        return;
    }

    FormOpenReply(r,m,&f->ObjNode, 0, pathname);

    PutMsg(&r->mcb);

    Free(r);
}
```

This function creates a new node in the directory tree and returns its name and a capability for it. The sequence of operations here is substantially the same for all dispatched procedures.

When do create is entered it is passed a `ServInfo` structure which contains the current state of the transaction, including the request message and a current position in the directory hierarchy. The `FnRc` field of the message is initialised to the subsystem code given in the `DispatchInfo` structure in preparation for generating a reply.

A GSP message contains a context directory name and an object, or target, name relative to it. The context directory will have been located and the capability in the message checked against it before do create is called. If an error is found during this process an error message will be generated by the server library automatically without calling the server code. In order to locate the target object the procedures `GetTargetDir` and `GetTargetObj` must be called. The first locates the parent directory of the target object and the second moves on from there to the object itself, if it exists. If any errors are detected by either of these procedures a `NULL` result is returned and the `EG` and `EC` fields of the message `FnRc` field are filled in.

The server library protects the data structures against concurrent update by keeping its current target object locked. On entry to a dispatched procedure the context directory is locked. When `GetTargetDir` returns, the target's parent directory is locked. When `GetTargetObj` returns either the target object is locked, or the parent directory remains locked if the target does not exist.

In do create the target object should not exist, so an error message is generated if it does. `ErrorMsg` sends an error message back to the sender of the request, it ORs the second argument with the value in the `FnRc` of the message. This allows the server to add an `EO` field to the error code generated by a server library function.

As `GetTargetDir` followed the path from the context directory, it updated the access mask stored in the capability in the message to reflect any changes in protection status. In `do_create` this final mask is checked to ensure that the client has write access to the directory, and hence can create the file. Following this we allocate a buffer for the reply message, call `CreateNode` to make the new file or directory, and finally call `FormOpenReply` to generate a reply message. The GSP functions `FG_Create`, `FG_Locate` and `FG_Open` all generate a special reply message containing the name and capability of an object, `FormOpenReply` builds this reply. Finally the reply message is sent back and the buffer freed.

The next procedure handles the Open operation.

```
static void do_open(ServInfo *servinfo)
{
   MCB *m = servinfo->m;
   MsgBuf *r;
   DirNode *d;
   File *f;
   IOCMsg2 *req = (IOCMsg2 *)(m->Control);
   Port reqport;
   byte *data = m->Data;
   char *pathname = servinfo->Pathname;

   /* find target's parent dir */
   d = (DirNode *)GetTargetDir(servinfo);

   if( d == NULL )
   {
      ErrorMsg(m,EO_Directory);
      return;
   }

   r = New(MsgBuf); /* get reply buffer */
```

```
if( r == NULL )
{
   ErrorMsg(m,EC_Error+EG_NoMemory);
   return;
}

/* find the target itself */
f = (File *)GetTargetObj(servinfo);

/* if file does not exist and O_Create set, create it */
if( f == NULL && (req->Arg.Mode & O_Create) )
{
   m->MsgHdr.FnRc = SS_Ram; /* clear error code */
   /* if file does not exist, see whether we are allowed to */
   /* create a file here. */
   unless( CheckMask(req->Common.Access.Access,AccMask_W) )
   {
      ErrorMsg(m,EC_Error+EG_Protected+EO_Directory);
      Free(r);
      return;
   }
   f = CreateNode(m, d, pathname);
}

if( f == NULL )
{
   /* does not exist: winge */
   ErrorMsg(m,EO_File);
   Free(r);
   return;
}

/* check we are allowed to perform the operation requested */
unless( CheckMask(req->Common.Access.Access,
   req->Arg.Mode&Flags_Mode) )
{
   ErrorMsg(m,EC_Error+EG_Protected+EO_File);
   Free(r);
   return;
}

/* generate reply message */
FormOpenReply( r, m, &f->objNode, Flags_Server|Flags_Closeable, pathname);

reqport = NewPort(); /* install message reply port */
r->mcb.MsgHdr.Reply = reqport;

PutMsg(&r->mcb);
Free(r);
```

The first half of do open is very similar to do create in locating and checking access to the target object, creating it if necessary, and generating a reply. The main difference is that the reply contains a new port which is to be used by the client to send requests to this process to access the file.

The next piece of code tests whether the object to be accessed is a directory. If it is, then the Server Library routine `DirServer` is called to handle all further requests.

```
if( f->ObjNode.Type == Type_Directory )
{
   /* directory - let ServLib handle it */
   DirServer(servinfo,m,reqport);
   FreePort(reqport);
   return;
}
```

If the `O_Truncate` bit is set then the file is truncated by calling `AdjustBuffers`. This server library routine is used to manage a list of Buffer structures and will be explained in more detail later.

```
/* if Truncate bit set - dispose of all data */
if( req->Arg.Mode & O_Truncate )
{
   f->Upb = 0;
   AdjustBuffers(&f->Buffers,0,0,FileMax);
}
```

We are now ready to enter a dispatching loop servicing requests from the client to access the file. Note that since the target may have been locked by `GetTargetDir` we unlock it before calling `GetMsg` to allow concurrent access to the file by several clients.

```
f->Users++; /* count an extra user */

UnLockTarget(servinfo);

forever
{
   word e;
   m->MsgHdr.Dest = reqport;
   m->Timeout = StreamTimeout; /* = 30 mins */
   m->Data = data;

   e = GetMsg(m);
```

```
if( e == EK_Timeout ) break; /* timeout - quit */

if( e < Err_Null ) continue; /* other errors - just loop */
```

At this point we have a valid request so we re-lock the file to ensure that this operation is atomic. We then decode the request and pass control to a suitable piece of handling code.

```
Wait(&f->ObjNode.Lock); /* lock file */

switch( m->MsgHdr.FnRc & FG_Mask )
{
case FG Read:
   FileRead(m,f);
   break;

case FG_Write:
   FileWrite(m,f);
   break;

case FG Close:
   /* close just frees the request port, unlocks & returns */
   if( m->MsgHdr.Reply != NullPort ) ErrorMsg(m,Err Null);
   FreePort(reqport);
   f->Users--;
   Signal(&f->ObjNode.Lock);
   return;

case FG GetSize:
   InitMCB(m,0,m->MsgHdr.Reply,NullPort,Err_Null);
   MarshalWord(m,f->Upb);
   PutMsg(m);
   break;

case FG Seek:
   FileSeek(m,f);
   break;

case FG_SetSize:
{
   /* change size of file - only truncates are allowed */
   word newsize = m->Control[0];
   if( newsize > f->Upb )
      ErrorMsg(m,EC_Error+EG Parameter+1);
   else
   {
      if( newsize < f->Upb ) f->Upb = newsize;
      AdjustBuffers(&f->Buffers,0,f->Upb,FileMax);
```

```
            InitMCB(m,0,m->MsgHdr.Reply,NullPort,Err_Null);
            MarshalWord(m,f->Upb);
            PutMsg(m);
         }
         break;
      }

      default:
         ErrorMsg(m,EC_Error+EG_FnCode+EO_File);
         break;
      }

      Signal(&f->ObjNode.Lock); /* free lock */
   }

   f->Users--;
   FreePort(reqport);
}
```

The final dispatched procedure implemented here is do delete, which removes the file or directory.

```
static void do_delete(ServInfo *servinfo)
{
   MCB *m = servinfo->m;
   File *f;
   IOCCommon *req = (IOCCommon *)(m->Control);

   f = (File *)GetTarget(servinfo);

   if( f == NULL )
   {
      ErrorMsg(m,EO_File);
      return;
   }

   unless( CheckMask(req->Access.Access,AccMask_D) )
   {
      ErrorMsg(m,EC_Error+EG_Protected+EO_File);
      return;
   }

   if( f->ObjNode.Type == Type_Directory &&
      ((DirNode *)f)->Nentries != 0 )
   {
      /* non-empty directory - complain */
      ErrorMsg(m,EC_Error+EG_Delete+EO_ Directory);
      return;
```

```
    }
    else if ( f->ObjNode.Type == Type_File )
    {
       if( f->Users != 0 )
       {
          /* file in use - complain */
          ErrorMsg(m,EC_Error+EG_InUse+EO_File);
          return;
       }
       f->Upb = 0;
       AdjustBuffers(&f->Buffers,0,0,FlleMax);
    }

    Unlink(&f->ObjNode,FALSE);
    Free(f);

    ErrorMsg(m,Err_Null);
}
```

Again the first half of do delete is fairly standard, the procedure `GetTarget` is simply a combination of `GetTargetDir` and `GetTargetObj` and is used when the parent directory of the object is not needed. The rest of the procedure simply determines whether the delete is allowed. Finally the delete is performed by `Unlink` which detaches the `ObjNode` from its parent directory and updates the directory entry count. The second argument to `Unlink` indicates whether the parent directory is locked, which in this case it is not.

The following three routines are called from do open to handle the read, write and seek requests.

```
void FiIeRead(MCB *m, File *f)
{
   ReadWrite *rw = (ReadWrite *)m->Control;

   word pos  = rw->Pos;
   word size = rw->Size;

   if( pos < 0 )
   {
      ErrorMsg(m,EC_Error|EG_Parameter|1);
      return;
   }

   if( size < 0 )
   {
      ErrorMsg(m,EC_Error|EG_Parameter|2);
      return;
```

```
    }

    if( pos == f->Upb )
    {
       m->MsgHdr.FnRc = ReadRc_EOF;
       ErrorMsg(m,0);
       return;
    }

    /* limit read to actual file size */
    if( pos + size > f->Upb ) rw->Size = f->Upb - pos;

    DoReed(m,GetReadBuffer,&f->Buffers);

    f->ObjNode.Dates.Access = GetDate();
}
```

A read request consists of a position, a size and a timeout (which is not used here). Most of this routine is concerned with checking that the position and size are within bounds.

The read request itself is handled by the system library routine `DoRead`. The arguments to this are the read request containing the (possibly updated) position and size, a buffer supply procedure and an argument to be passed to this procedure. In `DoRead`, whenever some data to be sent to the client is needed, the buffer supply procedure is called with the file position required and the argument supplied to `DoRead`. The result of this procedure is a pointer to a Buffer structure containing the data at the given file position. The procedure used here, `GetReadBuffer`, is supplied by the server library and works in conjunction with `AdjustBuffers` to manage a Buffer list.

```
void FileWrite(MCB *m, File *f)
{
    ReadWrite *rw = (ReadWrite *)m->Control;
    word pos  = rw->Pos;
    word size = rw->Size;

    if( pos > f->Upb )
    {
       ErrorMsg(m,EC_Error|EG_Parameter|1);
       return;
    }

    f->Upb = (f->Upb>(pos+size))?f->Upb:pos+size;

    /* re-arrange the buffers in the File */
    if( !AdjustBuffers(&f->Buffers,0,f->Upb,FileMax) )
```

```
    {
       f->Upb = ((Buffer *)f->Buffers.Tail)->Pos+FileMax;
       ErrorMsg(m,EC_Error|EG_NoMemory);
       return;
    }

    DoWrite(m,GetWriteBuffer,&f->Buffers);

    f->ObjMode.Dates.Modified = GetDate();
    f->ObjNode.Dates.Access   = GetDate();
 }
```

`FileWrite` handles write requests. Like `FileRead` the first part of this routine is concerned with checking and adjusting limits. The call to `AdjustBuffers` is used to add any extra Buffers needed to the end of the Buffer list. As implied already, `AdjustBuffers` is used to implement a simple list of data buffers. The first argument to `AdjustBuffers` is the address of a list of Buffer structures, the last argument is the size of the buffers. The remaining two arguments are the position of the last byte in the buffer list and the position of the last+1 byte. `AdjustBuffers` will add and remove buffers on the list to provide buffering for all the bytes between these two values. In this server the buffer list always starts at zero, but any value less than or equal to the buffer end position is permitted, allowing servers to maintain a moving buffer window.

The last routine is support for seek requests. Since the position is always sent with a read or write request this does nothing except re-calculate the new file position.

```
 void FileSeek(MCB *m, File *f)
 {
    SeekRequest *req = (SeekRequest *)m->Control;
    word curpos = req->CurPos;
    word mode   = req->Mode;
    word newpos = req->NewPos;

    switch( mode )
    {
       case S_Beginning: break;
       case S_Relative:  newpos += curpos; break;
       case S_End:       newpos += f->Upb; break;
    }

    if( newpos > f->Upb ) newpos = f->Upb;
    elif( newpos < 0 ) newpos = 0;

    InitMCB(m,D,m->MsgHdr.Reply,NullPort,Err_Null);
```

```
    MarshalWord(m,newpos);

    PutMsg(m);
}
```

4 Additional Features

In addition to the routines described above the server library also contains the following procedures (you are referred to `servlib.h` for the argument templates):

`GetContext(servinfo)`

This is used by the dispatcher to locate and check the context object of a GSP message, it is of little use to the general user unless you want to write your own dispatcher.

`pathcat(path,name)`

Concatenates the second string onto the first separating them with a */*.

`addint(name,val)`

Adds the string corresponding to the decimal value of its second argument to the string given as the first. It is not possible to add a value of zero with this routine.

`name = objname(path)`

Returns a pointer to the last component of the given pathname.

`obj = Lookup(dir,name,dirlocked)`

Searches the given directory for the named entry. If the third argument is false the directory will be locked during the search.

`MarshalInfo(mcb,objnode)`

Initialises the given MCB with a reply suitable to be returned as a result of the objectinfo request on the given object.

`newmask = UpdMask(oldmask,matrix)`

Returns a new access mask which consists of an old access mask updated by the given access matrix.

`success = GetAccess(cap,key)`

The given capability is decrypted with the key and, if valid, its cleartext

access mask ANDed with the encrypted mask. If the capability does not decrypt the procedure returns FALSE, otherwise it returns TRUE.

`Crypt(encrypt,key,data,size)`

The data buffer described by the last two arguments are (en/de)crypted with the given key.

`key = NewKey()`

A new encryption key is returned.

`NewCap(&cap,obj,mask)`

A new capability for the given object is created containing the given access mask.

`v = ServMalloc(size)`

This is used for memory allocation in all servers and `servlib.h` redefines `Malloc` to this routine. The server library keeps a safety net memory block which is freed if a memory allocation ever fails, this is implemented in `ServMalloc`. This allows servers to report memory allocation failure but to retain sufficient memory to allow a tidy cleanup to be implemented.

5 Converting from V1.0 to V1.1 Servers

A number of changes have been made to the server library in the upgrade from V1.0 to V1.1.

The main change is the locking of objects by `GetTargetDir` and `GetTargetObj`. In most cases this is transparent, except that a call to `UnlockTarget` must be added before the `do_open` dispatch loop. This has also caused the argument spec of a number of procedures to change, specifically `Insert`, `Unlink`, `Lookup` and `DirServer`, which will all be reported by the compiler.

Additionally the layout of the `DispatchInfo` structure has changed and some field names in `ServInfo` have altered. The original procedure `crypt` has been replaced by `Crypt`.