A Helios Standalone Server

Perihelion Software Technical Report No. 7

Bart Veer

May 1989

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

Contents

1 A Helios Standalone Server

In a client-server based system such as Helios, it is important to be able to create servers fairly easily. To help the programmer to do this, Helios provides a server library which is described in detail in technical report number 8. This server library is based on the assumption that the associated directory structure will be held in the tranputers' memory at all times. Whilst this is valid for some servers, like the processor manager, the loader and the RAM disc, it is not valid for others, such as a remote filing system, where the directory structure is held permanently on disc. This technical note describes how to write a server without using the server library. It uses extracts of code from existing stand-alone servers to illustrate many of the points, but the reader should note that the code fragments do not form a complete server.

All Helios servers must install themselves in the name table of the processor on which they are running. Once they are in the name table, Helios can locate the server from anywhere else in the network. To install itself, the server should `Locate()` the current name table and `Create()` a new entry in it, the new entry being of `Type_Name`. The following code may be used to perform this.

```
PRIVATE Object *NewName(string name, Port port, word matrix)
{ Nameinfo info;
  Object   *root = Locate(Null(Object), "/"), *NTE;

  info.Port     = port;
  info.Flags    = Flags_StripName;
  info.Matrix   = matrix;
  info.LoadData = NULL; /* Not used at present */

  NTE = Create(root, name, Type_Name,
               sizeof(NameInfo), (byte *)&info);
  Close(root);
  return(NTE);
}
```

The `NewName()` routine is given a name for the server, "`filesys`", a message port, and a protection matrix The protection matrix is usually either `DefDirMatrix` or `DefFileMatrix`, defined in the header `protect.h`, and may be used to limit access to the entire server if desired. The routine creates the entry and returns it, allowing the server to delete its entry from the name table at some future stage by a call `Delete(NTE, Null(char))`. This should be done as a tidy-up if the server exits.

Once the server is installed in the name table clients can access it using the

system library calls, directly or indirectly. These system calls are converted to messages sent to the message port associated with the name table entry. New message ports can be obtained and freed using the `NewPort()` and `FreePort()` calls. The following messages may be generated by the system library: Open, Create, Locate, ObjectInfo, ServerInfo, Delete, Rename, Link, Protect, SetDate, Refine, and CloseObj. The server should provide handler routines for all of these messages, even though some will just return error messages. The work involved is greatly simplified if the handler routines are held in a table.

```
#include <syslib.h>
#include <gsp.h>
#include <root.h>
#include <servlib.h>
#include <sem.h>
#include <nonansi.h>
#include <string.h>
#include <codes.h>

typedef struct Device {
   Port      Port;
   char      *Name;
   VoidFnPtr Handlers[12];
} Device;

PRIVATE void InvalidFun(MCB *, string);

PRIVATE void Drive_Open      (MCB *, string);
PRIVATE void File_Open       (MCB *, string, string, WORD opermode),
PRIVATE void Dir_Open        (MCB *, string, string);
PRIVATE void Drive_Locate    (MCB *, string);
PRIVATE void Drive_Create    (MCB *, string);
PRIVATE void Drive_ObjInfo   (MCB *, string);
PRIVATE void Drive_ServerInfo(MCB *, string);
PRIVATE void Drive_Delete    (MCB *, string);
PRIVATE void Drive_Rename    (MCB *, string);
#define Drive_Link           InvalidFun
#define Drive_Protect        InvalidFun
PRIVATE void Drive_SetDate   (MCB *, string);
#define Drive_Refine         InvalidFun
#define Drive_CloseObj       NullFn

PRIVATE Device server = {
   NullPort,
   Null(char),
   {   Drive_Open,
       Drive_Create,
       Drive_Locate,
```

```
        Drive_ObjInfo,
        Drive_ServerInfo,
        Drive_Delete,
        Drive_Rename,
        Drive_Link,
        Drive_Protect,
        Drive_SetDate,
        Drive_Refine,
        Drive_CloseObj
    }
};
```

The name and port for the server must be filled, and the server can then start accepting messages and forking off worker processes to deal with the requests. The routine `GSPServer` below, is in an infinite loop allocating a message buffer if possible, receiving a request, and forking off a `GSPWorker` process to deal with that request. The GSP Worker checks that the request is a valid one and calls a handler routine to deal with it. It is up to the handler routine to free the message buffer once it is finished.

```
PRIVATE void GSPServer(Device *Device)
{ Message *msg;

  forever
   { if ((msg = (Message *) Malloc(sizeof(Message))) eq Null(Message))
       { Delay(OneSec * 5); continue; }

     msg->mcb.MsgHdr.Dest = Device->Port;
     msg->mcb.Timeout     = OneSec * 30 * 60;
     msg->mcb.Control     = &(msg->control[0]);
     msg->mcb.Data        = &(msg->data[0]);

     lab1:
     while ( GetMsg(&(msg->mcb)) eq EK_Timeout);

     unless( Fork(Stacksize, GSPWorker, 8, Device, &(msg->mcb)) )
      { SendError(&(msg->mcb), EC_Error + SS_IOProc + EG_NoMemory +
                  EO_Memory, preserve);
        goto lab1;
      }
   }
}

PRIVATE void GSPWorker(Device *Device, MCB  mcb)
{ WORD      fn = mcb->MsgHdr.FnRc;
  VoidFnPtr fun;
  string    fullname;
```

```
  if ((fn & FC_Mask) ne FC_GSP)
   { SendError(mcb, EC_Error + SS_IOProc + EG_FnCode + EO_Message,
               release);
     return;
   }

  if (fn eq 0) /* ReplyOK message ? */
    return;

  fn &= FG_Mask;
  if ( (fn < FG_Open) || (fn > FG_CloseObj))
   { SendError(mcb, EC_Error + SS_IOProc + EG_FnCode + EO_Message,
               release);
     return;
   }

  if ((fullname = GetFullName(&(Device->Name[0], mcb)) eq NULL)
     return;

  fun = Device->Handlers[(fn - FG_Open) >> FG_Shift);
  (*fun)(mcb, fullname);

  Free(fullname);
}
```

To facilitate returning suitable messages there are routines `SendError()`, `InvalidFun`, and `Return()`, which will free the message buffer if there is no more use for it.

```
PRIVATE void InvalidFun(MCB *mcb, string fullname)
{ SendError(mcb, EC_Error + SS_IOProc + EG_WrongFn + EO_Server,
            release);
  fullname = fullname;
}

PRIVATE void SendError(MCB *mcb, WORD FnRc, WORD Preserve)
{ if (mcb->MsgMdr.Reply eq NullPort) return;
  *((int *) mcb) = 0;
  mcb->MsgHdr.Dest  = mcb->MsgHdr.Reply;
  mcb->MsgHdr.Reply = NullPort;
  mcb->MsgHdr.FnRc  = FnRc;
  mcb->Timeout      = 5 * OneSec;
  (void) PutMsg(mcb);
  if (Preserve eq release)
    Free(mcb);
}
```

```
PRIVATE void Return(MCB *mcb, WORD FnRc, WORD ContSize,
                    WORD DataSize, WORD Preserve)
{ if (mcb->MsgHdr.Reply eq NullPort) return;
  mcb->MsgHdr.Flags    = 0;
  mcb->MsgHdr.ContSize = ContSize;
  mcb >MsgHdr.DataSize = DataSize;
  mob->MsgHdr.Dest     = mcb->MsgHdr.Reply;
  mcb->MsgHdr.Reply    = NullPort;
  mcb->MsgHdr.FnRc     = FnRc;
  mcb->Timeout         = 5 * OneSec;
  (void) PutMsg(mcb);
  if (Preserve eq release)
     Free(mcb);
}
```

All the messages sent directly to a server will include the full name of the object that the client is trying to access, but extracting this from the message is non-trivial. The following routine deals with all the various combinations, allocating a new buffer to hold the complete name. This buffer must be freed by the GSPWorker process.

```
PRIVATE String GetFullName(string DeviceName, MCS *mcb)
{ BYTE *data  = mcb->Data;
  IOCCommon *common = (IOCCommon *) mcb->Control;
  int context = common->Context;
  int name    = common->Name;
  int next    = common->Next;
  string NewName, tmp;
  string dest = (string) Malloc(Name_Max);

  if (dest eq Null(char))
   { SendError(mcb, EC_Error + SS_IOProc + EG_NoMemory + EO_Server,
               release);
     return(Null(char));
   }
  else
   NewName = dest;

  for (tmp = &(DeviceName[0]); *tmp ne '\0'; )
    *dest++ = *tmp++;
  *dest++ = '/';

  for ( ; data[next] ne '/' && data[next] ne '\0'; next++)
    *dest++ = data[next];

  if (data[next] eq '/')
    for ( ; data[next] ne '\0'; next++)
      *dest++ = data[next];
```

```
  if (name eq -1) goto finished;

  if ( ((next < name) && (context < name)) ||
       ((next > name) && (context > name)) )
   { *dest++ = '/';
     for ( ; data[name] ne '\0'; name++)
       *dest++ = data[name];
   }

finished:

  if (*(--dest) ne '/') dest++; /* Get rid of any trailing '/' */
  *dest = '\0';

  if (!flatten(NewName))
   { Free(NewName); return(Null(char)); }

  return(NewName);
}

PRIVATE WORD flatten(string name)
{ char *source = name, *dest = name;
  int entries = 0;

  while(*source ne '\0')
   { if (*source eq '.')
       { source++;
         if   (*source eq '/') { source++; continue; }
         elif (*source eq '\0')
           { if (entries < 1) return(FALSE);
             dest--; break;
           }
         elif (*source eq '.')
           { source++;
             if (*source eq '/' || *source eq '\0')
               {
                 if (entries <= 1) return(FALSE);
                 dest--; dest--; while (*dest ne '/') dest--;
                 if (*source ne '\0')
                  { dest++; source++; }
                 entries--; continue;
               }
             else
               { *dest++ = '.'; *dent++ = '.'; }
           }
         else *dest++  = '.';
       }
```

```
      while (*source ne '/' && *source ne '\0') *lest++ = *source++;
      if (*source ne '\0')
        { *dest++ = '/'; source++;
          while (*source eq '/') source*+; /* This gets around a bug */
          entries++;                       /* in convert_name        */
        }
    }

  *dest = '\0';

  return(TRUE);
}
```

The code so far has been server-independent. Next we consider code to implement a particular server. Assume that the transputer can access a remote MSDOS filing system using the following routines. The exact means of accessing this filing system is irrelevant.

```
typedef int filedes;

typedef struct FileStream {
   WORD    pos;
   filedes fildes;
} FileStream;

typedef struct DirStream {
   WORD    number;
   WORD    offset;
   BYTE    entries[1];
} DirStream;

PRIVATE WORD exists_obj(STRING localname);
PRIVATE filedes open_file(STRING localname. WORD openmode);
PRIVATE WORD seek_in_file(FileStream *stream, WORD mode, WORD NewPos);
PRIVATE WORD read_from_file(FileStreem *stream, BYTE *buffer,
                            WORD amount);
PRIVATE WORD write_to_file(FileStream *stream, BYTE *buffer,
                           WORD amount);
PRIVATE void close_file(FileStream *stream);
PRIVATE DirStream *read_dir(STRING localname);
PRIVATE WORD create_object(STRING localname, WORD type);
PRIVATE WORD get_file_info(STRING localname, WORD *sizeptr,
                           Date *dateptr);
PRIVATE WORD delete_object(STRING localname, WORD exists);
PRIVATE WORD rename_file(STRING fromname, STRING toname);
PRIVATE WORD drive_statistics(STRING localname, WORD *sizeptr,
                              WORD *availptr);
PRIVATE WORD change_date(STRING localname);
```

The remote filing system works using MSDOS file names rather than Helios ones, so it is necessary to provide a name conversion routine to convert between the two, as follows.

```
PRIVATE char *GetLocalName(string HeliosName)
{ string tempptr, destptr;
  string local_name = (char *) Malloc(Name_Max);

  if (local_name eq Null(char))
    return(Null(char));

  for ( destptr = local_name, tempptr = HeliosName;
   (*tempptr ne '/') && (*tempptr ne '\0'); tempptr++);

  if (*tempptr eq '\0')
    { strcpy(local_name, HeliosName);
      strcat(local_name, ":");
      return(local_name);
    }
  else
    { *tempptr = '\0';
      strcpy(local_name, HeliosName);
      strcat(local_name, ":");
      *tempptr++ = '/';
    }

  for (destptr = &(local_name[strlen(local_name)]);
       (*tempptr ne '\0'); )
   { for (*destptr++ = '\\';
    (*tempptr ne '\0') && (*tempptr ne '/'); )
       *destptr++ = *tempptr++;
     if (*tempptr eq '/') tempptr++;
   }

  *destptr = '\0';

  return(local_name);
}
```

We can now consider a typical handler routine for one of the possible requests, `Drive_Locate()`.

```
PRIVATE void Drive_Locate(MCB *mcb, string fullname)
{ string localname = GetLocalName(fullname);
  WORD      exists;
```

```
  if (localname eq Null(char))
   {
     SendError(mcb, EC_Error + SS_IOProc + EG_NoMemory + EO_Server,
               release);
     return;
   }

  exists = exists_obj(localname);

  if (exists eq File_t)
   SendOpenReply(mcb, fullname, Type_File, 0, NullPort);
  elif (exists eq Dir_t)
   SendOpenReply(mcb, fullname, Type_Directory, 0, NullPort);
  else
   SendError(mcb, EC_Warn + SS_IOProc + EG_Unknown + EO_File,
             release);

  Free(localname);
}
```

`Drive_Locate()` extracts the local name given the Helios name. For example, if the Hellos name is "`c`" then the MSDOS name is "`c:`", and if the Helios name is "`c/helios/bin/ls`" then the local name is "`c:\helios\bin\ls`". Given this local name, `Drive_Locate()` calls one of the routines used to access the actual filing system, to determine whether the object exists and if so what it is. If the object does not exist the handler routine sends back an error message, releasing the memory allocated for the message, freeing the memory used to hold the local name, and returning to `GSPWorker()` above. `GSPWorker()` frees the memory used to hold the Helios name and returns, causing the worker process to terminate. Note that all the handler routines are called in separate processes, allowing different clients to access the server at the same time. It may be necessary to perform some locking inside the server using semaphores.

Successful replies to Locate, Create and Open requests must include a special data structure in the reply. This includes such details as a capability for the object and the full pathname for the object, allowing future accesses to the object to be faster. The routine `SendOpenReply()` constructs such a reply.

```
PRIVATE void SendOpenReply(MCB *mcb, string name, WORD type,
                           WORD flags, Port Reply)
{ IOCReply1 *reply = (IOCReply1 *) mcb->Control;
  if (mcb->MsgHdr.Reply eq NullPort) return;
  reply->Type     = type;
  reply->Flags    = flags;
```

```
  mcb->Control[2] = -1;
  mcb->Control[3] = -1;
  reply->Pathname = 0;
  reply->Object   = 0;
  MachineName(mcb->Data);
  strcat(mcb->Data, "/");
  strcat(mcb->Data, name);

  mcb->MsgHdr.Flags    = 0;
  mcb->MsgHdr.ContSize = sizeof(IOCReply1) / sizeof(WORD);
  mcb->MsgHdr.DataSize = strlen(mcb->Data) + 1;
  mcb->MsgHdr.Dest     = mcb->MsgHdr.Reply;
  mcb->MsgHdr.Reply    = Reply;
  mcb->MsgHdr.FnRc     = ReplyOK;
  mcb->Timeout         = 5 * OneSec;
  (void) PutMsg(mcb);

  if (Reply eq NullPort)
    Free(mcb);
}
```

Many of the other handler routines are also quite simple. `Drive_Create()` is used to create a new file or directory, or to truncate an existing file to zero length. It does not open a stream to the object. `Drive_Delete()` is used to delete a file or an empty subdirectory. The other handler routines tend to be fairly simple as well.

```
PRIVATE void Drive_Create(MCB *mcb, string fullname)
{ string localname = GetLocalName(fullname);
  IOCCreate *info  = (IOCCreate *) mcb->Control;
  WORD type;

  type = info->Type;

  if (localname eq Null(char))
   { SendError(mcb, EC_Warn + SS_IOProc + EG_NoMemory + EO_Server,
               release);
     return;
   }

  if ((type ne Type_File) && (type ne Type_Directory))
   { SendError(mcb, EC_Error + SS_IOProc + EG_Create + EO_Object,
               release);
     Free(localname);
     return;
   }
```

```
  if (!create_object(localname, type))
     SendError(mcb, EC_Error + SS_IOProc + EG_Create +
        ((type eq Type_File) ? EO_File : ED_Directory), release);
  else
     SendOpenReply(mcb, fullname, type, 0, NullPort);

  Free(localname);
}

PRIVATE void Drive_Delete(MCB *mcb, string fullname)
{ string localname = GetLocalName(fullname);
  WORD exists;

  if (localname eq Null(char))
   { SendError(mcb, EC_Error + SS_IOProc + EG_NoMemory + EO_Server,
               release);
     return;
   }

  if ((exists = exists_obj(localname)) eq Nothing_t)
   { SendError(mcb, EC_Error + SS_IOProc + EG_Unknown + EO_File,
               release);
     Free(localname);
     return;
   }

  if (!delete_object(localname, exists))
   SendError(mcb, EC_Error + SS_IOProc + EG_Delete +
            (exists eq Dir_t) ? EO_Directory : EO_File, release);
  else
   Return(mcb, ReplyOK, 0, 0, release);

  Free(localname);
}
```

The final point to consider is open streams, illustrated below by code to handle opening, closing, and reading files. Following a successful open, the server should include a new message port in its reply. This message port will be used for stream requests: Read, Write, GetSize, SetSize, Close, Seek, GetInfo, SetInfo, EnableEvents, Acknowledge, and NegAcknowledge. `File_Open()`, which is running as a separate process, waits for these messages and calls other handler routines to do the reading and writing. The system library will lock the open stream at the client side, so that the stream will only ever be sent one request at a time and there is no need to fork off worker processes. If no message arrives for the stream for 30 minutes it is assumed that the client has died without successfully tidying up, and the stream will go away automatically, closing the file in the process and doing

all the tidy-ups.

```
PRIVATE void Drive_Open(MCB *mcb, string fullname)
{ string  localname = GetLocalName(fullname);
  WORD    exists;
  WORD    openmode;
  IOCMsg2 *msg = (IOCMsg2 *) mcb->Control;

  if (localname eq Null(char))
   { SendError(mcb, EC_Error + SS_IOProc + EG_NoMemory + EO_Server,
               release);
     return;
   }

  openmode = msg->Arg.Mode;

  exists = exists_obj(localname);

  if (exists eq File_t)
   File_Open(mcb, fullname, localname, opennode & O_Mask);
  elif (exists eq Dir_t)
   { if ( ((openmode & O_Mask) eq O_ReadOnly) ||
     ((openmode & O_Mask) eq O_ReadWrite) )
      Dir_Open(mcb, fullname, localname);
     else
      SendError(mcb, EC_Error + SS_IOProc + EG_WrongFn + EO_Directory,
                release);
   }
  else
   { if ((openmode & O_Create) eq 0)
      SendError(mcb, EC_Warn + SS_IOProc + EG_Unknown + EO_File,
                release);
     elif (!create_object(localname, Type_File))
      SendError(mcb, EC_Error + SS_IOProc + EG_Creste + EO_File,
                release);
     else
      File_Open(mcb, fullname, localname, opermode & O_Mask);
   }

  Free(localname);
}

PRIVATE void File_Open(MCB *mcb, string fullname, string localname,
                       WORD openmode)
{ Port       StreamPort;
  FileStream stream;
  BYTE       *data = mcb->Data;
```

```
if ((StreamPort = NewPort()) eq NullPort)
 { SendError(mcb, EC_Warn + SS_IOProc + EG_Congested + EO_Port,
             release);
   return;
 }

stream.fildes = open_file(localname, openmode);

if (stream.fildes eq -1)
 { SendError(mcb, EC_Error + SS_IOProc + EG_Unknown + EO_File,
             release);
   return;
 }

SendOpenReply(mcb, fullname, Type_File,
              Flags_Closeable + Flags_MSdos, StreamPort);

stream.pos = 0;
            /* The stream is now open */
forever
 { WORD errcode;
   mcb->MsgHdr.Dest = StreamPort;
   mcb->Timeout     = StreamTimeout;
   mcb->Data        = data;

   if ((errcode = GetMsg(mcb)) eq EK_Timeout)
    { close_file(&stream);
      Free(mcb); FreePort(StreamPort);
      break;
    }

   if (errcode < Err_Null)
      continue;

   if ((errcode & FC_Mask) ne FC_GSP)
    { SendError(mcb, EC_Error + SS_IOProc + EG_WrongFn + EO_Stream,
                preserve);
      continue;
    }

   switch ( errcode & FG_Mask )
   { case FG_Read  : File_Read(mcb, &stream); break;

     case FG Write   : File_Write(mcb, &stream); break;

     case FG_Close   : File_Close(mcb, &stream);
                       Free(mcb);
                       FreePort(StreamPort);
                       return;
```

```
        case FG_Seek    : File_Seek(mcb, &stream); break;

        case FG_GetSize : File_GetSite(mcb, &stream);
                          break;

        case FG_SetSize :
        case FG_GetInfo :
        case FG_SetInfo :
        case FG_EnableEvents   :
        case FG_Acknowledge    :
        case FG_NegAcknowledge :
        default         : SendError(mcb, EC_Warn + SS_IOProc + EG_WrongFn +
                                    EO_Stream, preserve);
                          break;
      }
    }
}
```

Close requests may be sent either by the client itself, in which case the client expects a reply, or they may be sent on behalf of the client by the system, for example, when the client is terminated abnormally. In the latter case the system does not expect a reply to the Close request.

```
PRIVATE void File_Close(MCB *mcb, FileStream *stream)
{ close_file(stream);

  if (mcb->MsgHdr.Reply ne NullPort)
    Return(mcb, ReplyOK, 0, 0, preserve);
}
```

To achieve a degree of fault-tolerance, most requests are repeatable. In particular, when reading data from a stream the file position is sent with the request. Hence the request is of the form read 5000 bytes at offset 8000, and even if a reply message is lost the system library can just send exactly the same request again. This may mean that an implicit seek within the open file is required. Once the file is at the right position the server can start reading from it and sending the data back to the client. The amount of data requested may be larger than can fit into a single message, so the server may have to send multiple replies to a single request. The reply code may be `ReadRc_EOF` if the end of the file is reached before the read is satisfied, `ReadRc_EOD` if the reply is the last one and all the data requested has been sent, and `ReadRc_More` if the reply is not the last one. To inform the kernel that more messages are coming, the `MsgHdr_Flags_preserve` flag must be set on every reply except the last one.

```
PRIVATE void File_Read(MCB *mcb, FileStream *stream)
{ Readwrite *readwrite = (Readwrite *) mcb->Control;
  WORD  read_so_far, to_read, read_this_time, seq = 0, temp;
  bool  eof = FLSE;
  BYTE *buffer;
  Port itsport = mcb->MsgHdr.Reply;

  if (readwrite->Pos ne stream->pos)
   if (seek_in_file(stream, Seek_start, readwrite->Pos) eq -1)
    { SendError(mcb, EC_Error + SS_IOProc + EG_Broken + EO_File,
                preserve);
      return;
    }

  if (readwrite->Size eq 0)
    { Return(mcb, ReadRC_EOD, 0, 0, preserve);
      return;
    }

  if ((buffer = (BYTE *) Malloc(Message_Limit)) eq Null(BYTE))
    { SendError(mcb, EC_Warn + SS_IOroc + EG_NoMemory + EO_Server,
                preserve);
      return;
    }

  for ( read_so_far = 0; (read_so_far < readwrite->Size) && !eof; )
    { to_read = ((readwrite->Size - read_so_far) > Message_Limit) ?
            Message_Limit : (readwrite->Size - read_so_far);

      read_this_time = read_from_file(stream, buffer, to_read);
      read_so_far += read_this_time;

      if (read_this_time < to_read) eof = TRUE;
      mcb->MsgHdr.Dest  = itsport;
      mcb->MsgHdr.Reply = NullPort;
      mcb->MsgHdr.Flags = (eof) ? 0 : MsgHdr_Flags_preserve;
      mcb->MsgHdr.FnRc  = seq + (eof ? ReadRc_EOF :
    (read_so_far >= readwrite->Size) ? ReadRc_EOD : ReadRc_More);
      seq += ReadRc_SegInc;
      mcb->MsgHdr.ContSize = 0;
      mcb->MsgHdr.DataSize = read_this_time;
      mcb->Data = buffer;
      temp = PutMsg(mcb);
    }

  Free(buffer);
}
```