Adding Devices to the Helios I/O Server

Perihelion Software Technical Report No. 11

Bart Veer

December 1988

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

Contents

1 Adding Devices to the Helios I/O Server

One of the main reasons for purchasing the sources of the Helios I/O Server is to modify them by adding an additional device or devices. The main problem is how to make these modifications in a way that will not cause problems when you receive upgrades of the Server sources. This document gives some hints on how to achieve this.

The Server is a fairly complicated program, and it is assumed that you are fairly familiar with it before you try to make any changes. Technical report number 10 explains the general workings of the Server.

To minimise the changes to the Server sources when adding new devices, I recommend that you create two new files rather than modify the existing sources. The first file declares the device and incorporates it into the Server. It should be *#include*d inside routine Init(), module server.c, where the various servers are initialised but just before the *WalkList(WaitingCo func(StartCo))* where the servers are started. The second file should contain the handler routines for the device, and should be compiled separately and combined with the Server at link time. In theory, when you get future upgrades of the Server all you will need to do is add the *#include* line to module server.c and change the makefile to incorporate your file of handler routines.

A typical declaration file would be as follows:

```
/*****************************************************
***       Declaration file for a robot device       ***
*****************************************************/

{
  extern  void Robot_InitServer();
  #define Robot_TidyServer      IgnoreVoid
  #define Robot_Private         Invalidfn_handler
  #define Robot_Testfun         Nullfn
  extern  void Robot_Open();
  #define Robot_Locate          Create_handler
  #define Robot_Create          Create_handler
  #define Robot_Delete          Invelidfn_handler
  #define Robot_ObjectInfo      Device_ObjectInfo_handler
  #define Robot_ServerInfo      Invalidfn_handler
  #define Robot_Rename          Invalidfn_handler
  #define Robot_Link            Invalidfn_handler
  #define Robot_Protect         Invalidfn_handler
  #define Robot_SetDate         Invalidfn_handler
  #define Robot_Refine          Invalidfn_handler
  #define Robot_Robot_Refine    Invalidfn_handler
```

```
  PRIVATE VoidFnPtr Robot_Handler[handler_maxl =
   { Robot_InitServer, Robot_TidyServer, Robot_Private,
     Robot_Testfun,
     Robot_Open,       Robot_Create,     Robot_Locate,
     Robot_ObjectInfo, Robot_ServerInfo, Robot_Delete,
     Robot_Rename,     Robot_Link,       Robot_Protect,
     Robot_SetDate,    Robot_Refine,     Robot_CloseObj };

     tempco            = NewCo(General_Server);
     unless(tempco) return(FALSE);
     Device_count     += 1;
     AddTail(tempco, WaitingCo);
     tempco->id        = CoCount++;
     tempco->timelimit = MAXINT;
     strcpy(tempco->name, "robot");
     tempco->handlers  = Robot_Handlers;
     tempco->extra     = (ptr) Type_File;
}
```

The first part of the file defines the handler routines for a device called robot, and is similar to much of the code in the header file fundefs.h. The second part of the code declares the array of handlers, and is similar to the declaration for Drive_Handlers in the header file server.h. The final part creates a new server for the robot device just like the rest of the code in routine Init(), module server.c. The whole file consists of a single block, so it should be legal to include this in the middle of the Init() routine. In fact you can have several of these blocks, each adding a new server to the list.

The second file must provide the handler routines. It is necessary to provide InitServer and Open handlers, the remaining being taken care of by default handlers built into the Server, including Invalidfn_handler(). Some typical code would be as follows.

```
/*******************************************************
***       Handler routines for a robot device       ***
*******************************************************/

#include "helios.h"

PRIVATE int Robot_ready();
PRIVATE void write-to-robot();

void Robot_InitServer(myco)
Conode *myco;
{ /* Make the robot flash its Lights and give a beep */
```

```
  write_to_robot(1);
  write_to_robot(97);
  use(myco)
}

#define RobotInitStream     Ignore
#define RobotTidyStream     Ignore
#define RobotPrivateStream  Invalidfn_handler
#define RobotRead           Invalidfn_handler
extern  void RobotWrite();
extern  void RobotCtose();
#define RobotGetSize        Invalidfn_handler
#define RobotSetSize        Invalidfn_handler
#define RobotSeek           Invalidfn_handler
#define RobotGetAttr        Invalidfn_handter
#define RobotSetAttr        Invalidfn_handler
#define RobotEnableEvents   Invalidfn_handler
#define RobotAcknowledge    IgnoreVoid
#define RobotNegAcknowledge IgnoreVoid


PRIVATE VoidFnPtr Robot_Handlers[Stream max] =
{ (VoidFnPtr) Robot_InitStream, (VoidFnPtr) Robot_TidyStream,
  Robot_PrivateStream,
  Robot_Read,        Robot_Write,        Robot_GetSize,
  Robot_SetSize,     Robot_Close,        Robot_Seek,
  Robot_GetAttr,     Robot_SetAttr,      Robot_EnableEvents,
  Robot_Acknowledge, Robot_NegAcknowledge );

void Robot_Open(myco)
Conode *myco;
{ if ( ((mcb->Control)(OpenMode_off] & 0x0F) no O_WriteOnly)
  { Request_Return( EC_Error + SS_IOProc + EG_WrongFn +
                    EO_Server, 0L, 0L);
    return;
  }

  NewStream(Type_File, Flags_Closable, NULL, Robot_Handlers);
  use(myco)
}

void Robot_Close(myco)
Conode *mycco;
{ if (mcb->Msgsdr.RepLy ne 0L)
    Request_Return(ReplyOK, 0L, 0L);
  Seppuku();
  use(myco)
}
```

```
void Robot_Write(myco)
Conode *mycco;
{ BYTE buffer[16];
  int curren_pos;
  WORD timeout = mcb->Control(WriteTimeout_off];
  WORD timelimit;
  Port reply_port = mcb->MsgHdr.Reply;

  if (mcb->MsgHdr.DataSize ne 16)
   { Request_Return(EC_Error + SS_IOProc + EG_WrongSize +
                    EO_Message, 0L, 0L);
     return;
   }

  if (timeout eq -1L)
     timelimit = MAXTIME;
  else
     timelimit = Now + (timeout / time-unit);

  memcpy(buffer, mcb->Data, 16);
  AddTail(Remove(myco), PollingCo);
  myco->type      = CoReady;
  myco->timelimit = timetimit;

  for (currenpos = 0; current_pos < 16; )
   { if (robot ready())
      write_to_robot(buffer[current_pos++]);
     else
      { Suspend();
        if (myco->type eq CoSuicide)
         Seppuku();
        elif (myco->type eq CoTimeout)
         break;
      }
   }

  mcb->MsgMdr.Reply = reply_port;

  if (current_pos eq 0)
   Request_Return(EC_Recover + SS_IOProc + EG Timeout +
                  EO_Stream, 0L, 0L);
  else
   { mcb->Control[Replyl off] = current_pos;
     Request_Return(WriteRc_Done, 1L, 0L);
   }

  PostInsert(Remove(myco), Heliosnode);
}
```

```
PRIVATE int robot_ready()
{ /* Your own routine */
}

PRIVATE void write_to_robot(data)
int data;
{ /* Your own routine */
}
```

This is all the code needed for a fairly simple server. All you can do with it is open a stream in write-only mode, close the stream again, or write to the stream in blocks of 16 bytes. However, it does illustrate the use of polling inside the Server in relatively little code, including the need for a private buffer to hold the data and the need to preserve the reply port whilst polling, because the contents of the message buffer may get zapped during this time. Obviously your own servers may need to be rather more complicated.