Porting the Helios I/O Server

Perihelion Software Technical Report No. 14

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 Porting the Helios I/O Server

This document is a brief guide to porting the Helios I/O Server onto a completely new hardware configuration. This task is performed relatively infrequently, and hence this document is of little or no interest to ordinary Helios users. It is assumed that the reader has access to the Server sources and to the related technical notes no. 10 and 11, and that he or she is reasonably familiar with them. Note that this document provides some guidelines only and is not a step-by-step guide. Anyone who wishes to purchase the server sources can do so by contacting Perihelion Software Ltd.

Porting the Server involves at least five separate stages:

1. setting up the configuration so that the bulk of the Server can be compiled on the target hardware.
2. writing the link input/output routines.
3. writing the terminal input/output routines. After this stage it should be possible to debug the transputer and thus test the link input/output routines.
4. adding support for the multi-threading used inside the Server.
5. adding the local routines needed to support file input/output.

After these stages, it should be possible to boot up the basic Helios system and run most programs. Hence this is a good time to ensure that the current version is robust, before attempting to support the more esoteric parts of the Server such as multi-tasking, RS232 supports, X-Window support and so on. All the various stages are described below.

Please note that the Server has gone through many iterations and is still undergoing continuous improvement. Hence it is likely that you will receive upgrades to the sources at some future stage and that you will need to merge your changes. When porting the Server it should be necessary to make few changes to the main Server sources, since most of the porting work involves local, machine-specific modules. I am happy to try to merge any changes you need to make to the main sources with the current system, to make upgrades easier and to make the Server more portable.

2 Configuring the Server Sources

The first step in porting the Server is to configure it for the hardware. This involves modifying two files: the header `defines.h` and the module `files.c`. In the header `defines.h` you should add an entry for your particular machine or correct the existing entry. Initially the Server should be configured for as basic a system as possible (that is, you should not support any of the devices, multi-tasking, the ANSI emulator and so forth). You should incorporate the debugger, because this is an important step in the porting process. In the module `files.c` you need to provide a set of macros or external function declarations for the low-level file input/output routines. I recommend that initially these should all be declared as external functions, leaving optimisation for a later stage.

Whilst configuring the system you should create a subdirectory for the local module(s), in keeping with the general organisation of the Server sources, and modify one of the existing makefiles as required to compile and link the Server. In theory you can now compile the Server, and get a list of undefined functions at link time. In most cases these undefined functions are local routines which are covered in the sections below, and I recommend looking at the existing set of local routines for, for example, the IBM PC or the Sun. Also, many of these functions may be `#defined` to library routines in the header `fundefs.h`, for particular configurations.

The Server sources compile fairly happily under a number of different C compilers and use many of the standard C library routines. However, C libraries tend to be relatively inefficient and hence lower level routines should be used in many cases for efficiency. The header file `barthdr` deals with many of the incompatibilities between different C compilers, and it may be necessary to modify it in order to avoid generating certain compiler warnings and so on.

3 The Link I/O Routines

The Server must be able to interact somehow with the root transputer. To achieve this you need to write a fairly small number of link input/output routines, which should be put into local modules. At the very least you need the following routines:

1. `int byte_from_link(byte *)` - read a single byte from the link, returning 0 for success or 1 for failure (!!!). If successful, the byte should be stored at the address specified. This should fail if no byte is ready for about 500 milliseconds.

2. `int byte_to_link(int data)` - send a single byte of data down the link, returning 0 for success or 1 for failure.

3. `int fetch_block(int count, byte *data, int timeout)` - read a block of data from the link in the timeout specified. The timeout is pretty well irrelevant, and is usually treated as the number of times you should go around a polling loop. The routine should return 0 for success, or the number of bytes it failed to fetch. Note that if it failed to fetch 0 of the expected number of bytes it succeeded.

4. `int send_block(int count, byte *data, int timeout)` - similar to the above but for sending data down the link.

5. `int rdrdy(void)` - check the link to see if there is any data ready, returning 0 if there is no data and non-0 otherwise.

6. `int wrrdy(void)` - check the link to see if the other side is ready to receive data. This routine is rarely used, and may always return success if it is difficult to implement.

7. `void init_link(void)` - this call is used to initialise or reset the link adapter. It may be called many times.

8. `void reset(void)` - this call is used to reset the root transputer, and possibly other bits of the transputer network as well depending on the hardware.

9. `void analyse(void)` - this call is used to analyse the root transputer. `Analyse()` involves asserting the analyse line on the transputer, a short delay, asserting the reset line, another delay, releasing the reset line, and then releasing the analyse line. The timing details may be found in Inmos documentation.

There are two approaches to implementing these link routines. On the Sun there is a particular device driver (for example, `/dev/MK0`, which can be opened, read from, written to and so on); `ioctl` calls are used for reset, analyse, and so on; hence the link routines can be written entirely in C. On an IBM PC or compatible the link routines interact directly with the link adapter chip, and hence they are written in assembler.

Given the link input/output routines described above it is possible to construct some higher level ones which are actually called by the Server, such as the message passing routines. These are written entirely in C, and may be found in, for example, the Sun or the PC local modules. However, if the target hardware is not little-endian there may be some further problems lurking in this area (for example, problems with byte pointers versus int

pointers). Note that it may be desirable to implement all of the routines including the higher level ones in assembler for efficiency, although the link I/O tends not to be a bottleneck.

4 The Terminal I/O Routines

In the absence of multiple windows or the ANSI terminal emulator, you should need just two routines to get terminal I/O up and running. The first routine is `void output(char *)`, which should just output the `'\0'` terminated string to the screen. Ideally it should accept the ANSI escape sequences, but this is not essential at this stage. The second routine is `int read_char_from_keyboard(int x)`, which is used to poll for keyboard input. The argument `x` should be ignored, and the routine should return -1 if no character is ready to be read or else it should return the character. Ideally the keyboard should use the ANSI sequences for the cursor and function keys and so on; plus the Helios extensions (documented in the Helios manual), and the routine should support the run-time debugging options such as CTRL-SHIFT-F10, but this is not essential at this stage.

5 Getting the Debugger Up and Running

It is now time to recompile and relink the Server. There will still be some missing functions for the coroutine library and for the file I/O routines. Dummy functions can be used for these for the time being. If there are any other routines missing (for example, memory allocation), these should be put in now; they must not be left as dummies because the debugger may depend on them.

Once the Server has been linked it can be run with the `-d` option, and you should have a working debugger. Amongst other things you should be able to peek and poke the transputer memory, boot the transputer and get a few messages back, and so on. The debugger is described in the chapter on the Server in the user part of the manual. An awful lot of things can go wrong at this stage, particularly with the link routines, and these should be fixed before going any further. Only when all the debugging routines seem to work reliably should you attempt to get the Server itself up.

6 Multithreading

As described in technical note no. 10, the Server depends on a simple form of multi-threading. There are two approaches to supporting this. First, you can produce a standard library consisting of the library calls `InitCo()`, `CreateCo()`, `DeleteCo()`, `WaitCo()`, `ResumeCo()`, and `CallCo()`. The Server sources include a 68000 version for the Sun and an 8086 version for the IBM PC, which can be used as the basis of this library. The Server actually works in terms of some higher level functions `InitColib()`, `TidyColib()`, `NewCo()`, `Suspend()`, and `Seppuku()`, and usually these are implemented in terms of the coroutine library. However, the second approach to supporting the multi-threading is to implement these higher level routines in terms of whatever is provided by the system, and compiling out (by `#ifs`) the current code.

7 The File Input/Output Routines

The final stage in porting the basic Server is to support the file input/output routines. This requires a number of local routines or macros such as `int object_exists(char *name)`, described in quite a bit of detail near the top of module `files.c`. There is little point in repeating the relevant information here.

Incidentally this is probably the right time to implement the full keyboard, especially the run-time debugging facilities. Finding out what is going on without a simple means of enabling and disabling debugging, or the ability to exit or reboot the system quickly, can be quite painful.

After putting in this support, the Server should be able to boot the transputer and start handling the GSP messages. Initially you will be lucky to get a few messages such as distributes searches for the helios server, but as you debug the local routines more and more should start to happen until eventually you are faced with a shell prompt and able to type in and run some commands. If you do not appear to be getting any messages at all the most likely problem is that the message structure in the header file `iomess.h` is incorrect for the byte ordering in use, and you should put debugging information into the `GetMsg` routine.

At this stage the most useful tool is a working Helios system with the debugging options enabled to allow comparison. Secondly, if you cannot get it to work this is the right time to disturb me and I may able to work out what is going wrong just on the basis of the messages being sent to and from, or the data arriving from the transputer. After a couple of hours/days of debug-

ging you should have a fairly robust albeit rather basic Server, and it may be time to incorporate some of the other options available. In order of priority these are likely to be multiple windows and ANSI emulation, multi-tasking, RS232 and other ports, and X-Windows support. I give brief outlines below of the work involved in supporting some of these, but for more information the reader should consult the current sources.

8 Windows and the ANSI Emulator

Unless your hardware supports the ANSI screen escape sequences already you will need to incorporate the ANSI emulator. This involves modifying the `defines.h` header, removing your `output()` routine from the local modules, adding some routines to the bottom of module `terminal.c` for such jobs as moving the cursor to a particular position on the screen, and recompiling. Once you have full ANSI emulation and once you support the ANSI sequences for cursor keys and the like you can run the Helios version of microEmacs. To test screen output generally, I have an ansitest program which you can use.

If your hardware supports multiple windows it is rather desirable that Helios can make use of them. This involves changing the appropriate configuration option in `defines.h` and adding some routines for creating and deleting windows. In addition, you have to modify `read_char_from_keyboard()`, because the argument will actually be the handle of the window: each window is polled separately for keyboard input. Even if the hardware does not support multiple windows it is very desirable to implement pseudo-windows, where each window is drawn on a shadow screen and only one is currently active. A key sequence, such as ALT-F1, is used to switch between shadow screens. The main implementation of this so far has been for the IBM PC.

At present the ANSI emulator will not work with real windows, particularly resizable ones, but the work involved should be relatively minor and I will do it on request.

9 Multitasking Support

Given a machine such as a Sun or a VAX, it is rather sad if it spends all its time in the Server's main polling loop waiting for the user to type something. Hence the Server contains code, compiled in by setting an option in the `defines.h` file, which will make it inform the local modules whenever it is waiting for input of some sort and whenever it stops waiting for this input; inside the main loop another local routine is called, which is allowed to

suspend the Server until one of the inputs is ready, until a message is waiting on the link, or until a timeout has expired. Because most input/output is performed with finite timeouts, the Server cannot be suspended indefinitely.

The relevant routines are `InitMultiwait()`, `TidyMultiwait()`, `AddMultiwait()`, `ClearMultiwait()`, and `Multiwait()`. The main implementation, experimental at the time of writing, is for the Sun. Note that this code is liable to change in the future when I get more time and suitable hardware.