The Helios I/O Server

Perihelion Software Technical Report No. 10

Bart Veer

November 1988

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

Contents

1 The Helios Input/Output Server

At the time of writing all implementations of Helios run on a transputer add-on to an existing computer such as an IBM PC or compatible or a Sun Workstation. Helios input/output is performed by server software running on the host computer. In most cases this server software is the Helios I/O Server. The Server is responsible for eliminating as much of the hardware dependency as possible, allowing the same version of Helios to run on all the different configurations unchanged.

The Server has to satisfy a number of different requirements. First, it has to be highly portable so that as little work as possible needs to be done to port Helios to a different hardware configuration. Second, it has to be extendible so that new devices can be added fairly easily. Third, it has to be efficient. Fourth, the code must be easy to understand and maintain since many different people may want to alter it. Finally it has to be robust.

This document attempts to describe the general structure of the Server. It is aimed mainly at people who have access to the Server sources and are trying to understand them before making modifications. Please note that the Server has gone through many iterations, and is still undergoing continuous improvement: this document refers to version 3.60 of the Server, but most of it is appropriate to earlier and later versions. There are two related technical notes: number 11 deals with adding new devices to the Server, and number 12 deals with porting the Server to new hardware.

The source code for the I/O server can be purchased from Perihelion Software Ltd. Please contact us for further details.

2 Introduction

The Server can be divided into three main components: hooting the transputer; debugging the transputer; and server mode. The first two are fairly straightforward, unlike the last.

The Server allows the host processor to emulate a node in the transputer network, usually called `/IO` or `/Cluster/IO`. As far as Helios is concerned this node is no different from any other: it behaves just like a transputer running Helios. In fact you can even try to run programs in it, though with little chance of success. Inside this network node there are a number of servers: there will be a server for every drive in the filing system; there will be a window or console server; there may be an RS232 server, and so on. Note the distinction: Server with a capital S refers to the entire program, which contains Helios servers for a number of devices. Client programs on

the transputer side of the network can interact with these servers and, in most cases, open streams to the corresponding devices.

Interaction between the Server and the transputer network occurs at two levels. First, the Helios kernel performs fairly regular handshaking between all links in the hope of detecting when a network node goes down, and the Server has to support this: the details of this handshaking are of very little interest to anybody. Secondly, client programs can interact with servers and open streams by sending Helios messages, usually by calling the system library directly or indirectly. Most of these messages satisfy the General Server Protocol or GSP, but for some servers and open streams additional private protocols may be used. These protocols merely specify the contents of the message, not the format of the message itself which is fixed.

The bulk of the Server is concerned with receiving Helios messages, working out which server or stream they are meant for, turning the GSP request into a local operation (for example, converting a Helios Read into an MS-DOS file read), and constructing and sending back the GSP reply. Most of this work is hardware independent: only the local operations and the transferring of messages differ between implementations. However, even though much of the code is concerned with encoding and decoding messages, most of the time is taken up by performing the physical input/output and transferring the messages.

3 The Server Sources

The Server sources, as distributed to some customers, are spread over a number of different directories. First there is the Serving directory, containing the header files `iohelios.h`, `iocodes.h`, `ioprot.h`, `iomess.h`, `iogsp.h`, `iosyslib.h`, `ioattrib.h`, `ioevents.h`, `ioconfig.h` and `ioaddon.h`. These header files define the protocol used between the Server and the transputer network: they have their equivalents `helios.h` etc on the transputer side. Given that the protocol has now steadied down into a stable state it is likely that these header files will be combined into a single `protocol.h` header.

Next there is the main Server directory, containing the bulk of the Server. There should be a number of subdirectories, typically `ibm` and `sun`, which contain the additional code needed to get the Server running on different configurations. For example, the `ibm` subdirectory should contain the files `pclocal.h` with additional header file information needed by the Server, `pclocal.c` and pcasm.asm containing the local routines needed by the Server, and `makefile.pc` and link.lnk to allow the Server to be remade on an IBM PC or compatible.

Each file in the Server directory has a particular purpose. The first is `defines.h`: when compiling the Server, the command used to compile each module should contain a `-DSUN` or `/DPC` or some similar directive, identifying the configuration. Inside `defines.h` this machine identifier is used to set a large number of manifest constants giving a fairly exact specification of the hardware configuration and determining what devices will be supported by the Server in this configuration. For example, the manifest RS232 supported is set to 1 or 0 indicating whether or not the Server should include an RS232 device server. Another important hardware dependency is the byte ordering: if the host machine is little-endian like the transputer everything is fairly simple, but if it is big-endian (for example, a 68000 the Server has to do a lot of byte swapping when sending and receiving messages); this also affects some of the data structures used.

The header file, `barthdr`, is used to specify some of the stranger macros I insist on using. In addition it takes care of much of the incompatibilities between different C compilers. For example, on a Sun the `tolower()` routine is not defined if the argument is not an upper case letter, but under Microsoft C it works for all arguments.

`Structs.h` contains some data structures used internally by the Server (for example, the linked lists, plus some manifests). `Fundefs.h` declares the routines shared between different modules. Some of this is configuration dependent; for instance, routine `swap()` is defined either as a function or as a null macro depending on the byte ordering. In addition many device routines are declared as invalid. It is illegal, for example, to delete the console device so `ConsoleDelete` is defined as an invalid function. `Server.h` declares most of the global variables. It also defines an array of device servers. This array is used extensively in the Server for efficiency. For example, when a server receives a delete request it can invoke the appropriate function in its table, which may be a real function or it may have been defined as a general purpose function of some sort (for example, one to handle inappropriate requests). The array also contains some extra functions to initialise and tidy servers, to check that the device is actually present before creating a server for it, and to handle private messages. Similar arrays are defined for streams.

The remaining header file, `debugopt.h`, is used to control some of the debugging facilities available within the Server, and is of little interest.

`Server.c` is the main source file. `Tload.c` is concerned with booting up the transputer and transferring Helios messages. `Files.c` takes care of the Helios side of file input/output. Similarly `terminal.c` is for terminal input/output (screen + keyboard) and `devices.c` is for other devices (RS232, mouse, and so on). `Cofuns.c` contains the linked list library and

the multi-threading library. `Debug.c` and `dbdecode.c` contain the code for the debugging mode. These sources are all described in more detail below.

4 Server Startup

Quite a bit of work is involved setting up the world before anything useful can be done. Most of this is done directly or indirectly by `main()`. There are some configuration dependent bits here such as ignoring the arguments when running the Server under GEM on a PC because they are not set up correctly. Then there is an all-important `setjmp()` to allow other initialisation routines a clean emergency exit. Then I process the arguments, and read in the configuration file.

The purpose of the configuration file is to control run-time rather than compile-time options. Since it can be changed by ordinary users it has to be simple. Hence configuration file entries are of the form X or X = Y. Some entries are compulsory, mainly to check that the file is still valid and has not been overwritten. Some entries are configuration-specific (that is, they are required by the local modules rather than by the main Server), and hence I have no way of knowing exactly what entries I may have to store. To get around this I just store all the entries in a linked list and provide interrogation routines `get_config()` and `get_int_config()` to allow the list to be examined. `get_config()` is similar in operation to `getenv()` under Unix, and since many configuration entries are numbers of some sort it is useful to provide an integer version of it.

After reading in the configuration file I call `initalise_devices()`. This routine lives up to its name: it has to do any one-off device initialisations such as taking over interrupt vectors. In a multiple-windowing system its main job is to set up the Server's own window, which used to display Server output, such as debugging messages, or warnings that, for example, a floppy disc is write protected. In addition, `initialise_devices()` usually calls a configuration-specific routine.

The next step is to load the bootstrap program, usually `nboot.i`. This program is guaranteed to be ≤ 256 bytes large, because of the way booting the transputer from a link works. Hence there is no harm in leaving it permanently loaded. Since it is possible to switch between server mode and debug mode fairly easily, I initialise the debugger part of the Server here and leave it initialised.

In theory I now have all the information I need to either boot up the transputer or to start the debugger. There is an infinite loop which does a restart for the various devices, and runs one of the two modes. On breaking out of

this loop, or if one of the initialisations failed and did a `longjmp()`, I tidy up the world in the reverse order of initialising it and `exit()`.

5 Booting Up the Transputer

The bootstrap process in module `tload.c` involves a number of separate stages. First, the boot program `nboot.i` and the system image have to be loaded off disc `nboot.i` is small so it is kept permanently loaded, but a system image is typically about 60K which takes up too much of fairly scarce memory on some configurations. Hence in server mode the system image is loaded off hard disc and the memory is freed again as soon as the transputer has been booted. However, in debug mode it is very useful to keep the image loaded because it contains many system identifiers.

Next I call `resetlnk()`. This is a no-op on most configurations, but sometimes the link adapter can be in a strange state and needs to be reset. Then I reset the transputer, to make sure that it is ready to receive a bootstrap. This is followed by sending the size of the bootstrap program down the link, which starts up the transputer boot-down-link hardware, and the bootstrap program itself. The Helios bootstrap, `nboot.i`, is a fairly clever program for its size and has a number of facilities such as clearing memory. Each option is invoked by sending a particular byte down the link, possibly followed by arguments. On machines with parity memory, notably the Meiko Computing Surface, the off-chip memory has to be cleared before it can be used.

After that little lot I am ready to send the Helios system image down the link, in several chunks because of possible problems in sending large amounts of data in one go. This system image contains the kernel, the system, server, and utility libraries, the processor manager and loader, and an initialisation program which usually starts up a shell. When the entire system image has been sent the bootstrap program transfers control to the kernel, which needs some additional configuration information before it can initialise itself. This configuration includes details such as the processor names, and the structure is defined in the header file `config.h`.

In theory Helios is now up and running, and will start sending some data. The first data is guaranteed to be a probe request, consisting of a byte `0x00F0` followed by another 11 bytes. In server mode, the infinite loop in `main()` will call `server_helios()` to receive this byte `0x00F0` and reply appropriately. In debug mode the debugger takes care of receiving this. Note that probe requests may be sent at any time, not just during start-up.

Module `tload.c` also contains the message interface between Helios and the

Server. During start-up a single message buffer is initialised, and the size of the data vector is controlled by the configuration file. For the sake of efficiency this message buffer is used by everything, so that I do not need to worry about copying the messages to other buffers etc. Also, the message buffer can account for much of the Server's memory usage, so having a single message buffer cuts down on that.

All message passing goes via Request State, and `Request_Return()`. `Request_Stat()` polls for a message with a short timeout, so that the Server is not suspended indefinitely. If `GetMsg()` returns `0x00F0`, the Server has received another probe request, which is handled as usual; otherwise it has received a Helios message. This may be part of an `IOdebug()` on the transputer side, in which case the function code is a particular magic number and the data vector contains a single character: there is no need to reply to such a message. Otherwise the message is for a server or an open stream, and I return it to a higher level to avoid problems with port tables and so forth in the kernel, the flag field in the message is cleared, and servers and streams have to set it if necessary. When a server or stream needs to send a message back to the transputer, it has to go via `Request_Return()`. There is a test for sending a message to NullPort, just in case. If message debugging is currently enabled, details of the message are output to the screen. If for some reason the Server is unable to send the message then something has gone seriously wrong: the transputer should have a link guardian running at high priority and waiting on the link, so the Server should always be able to send a message within a fairly short time.

6 Multi-Threading and the Main Loop

Once the transputer has been booted and the kernel is active, `main()` calls routine `Server()` in module `server.c`. This initialises server mode, runs `MainLoop()` until either the user or the transputer has specified that the Server should exit server mode, and then tidies up server mode.

The main problem in understanding the Server is its use of multi-threading. The reason for this is essentially the same as for using multiple threads or processes when programming on a transputer: it makes programming a lot easier. At the time of writing all the implementations of the Server use coroutines to implement multi-threading: this has the advantage that only one thread is active at any one time, which means that I can use global data including a global message buffer without synchronising via semaphores etc However, there is no particular reason why coroutines must be used: if the system provides some cheap multi-threading facility already, which allows synchronisation so that only one thread is active at any one time, then this

can be used perfectly safely instead.

The Server uses separate threads for every device server and for every open stream. These threads are held in one of two linked lists: if the server or stream is currently waiting for a message from the transputer (for example, the next request), then it is held in `WaitingCo`; otherwise it is held in `PollingCo`, and any attempt by the transputer to send messages to it will fail because the device is already busy. The `MainLoop` of the Server performs the following: wake up any threads held in `PollingCo` to let them check whether they should be doing anything; get a message if the transputer is trying to send one; if this message is for a thread held in `WaitingCo` that thread gets woken up; otherwise an error message should be sent. There is a one to one correspondence between message ports on the Server side and threads.

Routine `Init()` initialises the multi-threading library by a call to `InitColib()`, and initialises the linked lists `WaitingCo` and `PollingCo`. It works out the location of the Helios directory in the host's filing system, usually via the configuration file. Then it creates threads for every device server supported by the configuration: these are all held in the table defined in header `server.h`. A new thread is created by a call to `NewCo()` in module `cofuns.c`, which returns a pointer to a `conode` structure defined in header `structs.h`. In addition, if the host has multiple drives in its filing system these are treated as separate devices and threads are started for all of them. Once all the threads have been created they are started up by calls to `StartCo()`, so that the various servers can initialise themselves. The Server is now ready to receive and process messages from the transputer network.

`TidyUp()` performs the opposite function to `Init()`. Its main job is to persuade all the threads in the Server to kill themselves, Whenever threads suspend themselves (and they continue running until they suspend themselves), as soon as they are woken up again they must check whether or not they are meant to stop or commit `seppuku()`. All that `TidyUp()` needs to do is tell all threads that they should stop and then wake them up.

All servers are started up in a routine `General_Server()` in module `server.c`. The argument to General Server is a pointer to the `conode` structure corresponding to that thread, which contains amongst other things a pointer to an array of handler functions. Hence when `General_Server()` gets woken up with an Open request from the transputer, it can call the appropriate handler function very easily. One of these handler functions is used to initialise the server, frequently a no-op. The header file `fundefs.h` defines which handler functions are no-ops or error handlers, and which ones do real work. The current thread suspends itself, and will get woken up either when it is told to die or when a message has arrived for it. If it is a valid

message then this message must contain a name, which is converted and stored in a static buffer. This saves having to convert the name in every handler function for every device. Then the handler function is invoked, and the thread suspends itself thus giving control back to `MainLoop()`.

In addition there is a routine `General_Stream()`, which is similar to `General_Server()` but for streams rather than servers. Streams get a different set of requests from servers and need a different set of handler routines. Also, if a stream is not used for a long time then it is assumed that the client who opened that stream has died and the stream thread dies off (unless there is a very good reason for not doing so). Note that it is very easy for servers to create new streams, by calling a `NewStream()` routine in module `server.c`.

7 A Typical Interaction

To explain the above a bit more dearly, I shall describe a typical example. Suppose that a program wishes to read the `/IO` directory. It sends an Open request to the server for IO. Next time `MainLoop()` calls `Request_Stat()`, this message is read from the link and stored in the main message buffer. `MainLoop()` looks at the destination port held in the message and discovers that it matches the thread identifier associated with the IO server. Hence this thread gets woken up, inside routine `General_Server()`. The thread looks at the message, discovers that it is a valid open request, converts the name to IO, and calls the appropriate handler function. From the header files `server.h` and `fundefs.h` we can determine that this function is in fact `IOProc_Open()`, which happens to be in module `server.c` as well. `IOProc_Open()` determines that the name held in the message is valid, that the client is not trying to do something silly, and then creates a new stream. The `NewStream()` routine creates a new thread, initialises its handler functions, and starts the new thread. This thread starts off in routine `General_Stream()`, which initialises the stream by calling a handler function. Then it sends a reply back to the client, and the stream suspends itself. This reactivates the parent thread (that is, the server for IO), which returns to `General_Server()` where this thread is suspended. Now `MainLoop(` is reactivated, and the Server is ready to receive a new message.

When the client gets back the reply message, it will have a message port for the stream just opened. Hence it can send Read requests to this stream. Again, these are accepted by `MainLoop()`, and result in waking up the thread inside `General_Stream()`. The Read request is handled by invoking the appropriate handler function, and eventually control returns to `MainLoop`. When the client sends a Close request the stream thread tidies up and commits `seppuku()`, thus removing itself from the linked lists `WaitingCo`

and `PollingCo` and preventing any further messages being sent to it.

As a slight complication, consider a Read request for the console device (that is, the client wants to know the next key pressed). When the Read request arrives it is quite likely that the user has not pressed a key, and the thread has to wait a while before it can send back a reply. Whilst waiting it cannot handle any new requests: for example, if it received another Read request and the user pressed a key, the stream would have the difficult job of working out which client should get the key. However, the other threads are still active and can handle additional requests coming in. When a thread needs to wait for something to happen before it can send a reply, it should move its `conode` from the `WaitingCo` list to the `PollingCo` list. This achieves two things: first, all threads in `PollingCo` are woken up regularly by the `MainLoop` so that they can check whether or not anything interesting has happened recently; second, threads held in `PollingCo` cannot receive new messages. A complication here is that under Helios most input/output is performed with timeouts, to achieve limited fault-tolerance. Hence threads involved in polling need to worry about timeouts as well as other events. This is done automatically by putting a time limit in the `conode` structure, which is checked by `MainLoop`.

8 The IOProcessor Device

The Server has to emulate a full Helios node running on the host machine. Hence it must be possible to list `/IO` to determine what servers are running on that particular machine. Also, when the network server extends the current network the Server node will be renamed (for example, from `/IO` to `/Cluster/IO`). Also, as an intelligent network node the Server has to support distributed search requests. Hence when a client on some remote transputer tries to access `/rs232/default` for the first time, this may initiate a search for a server called rs232 which gets sent to all network nodes, and it is the IOProc server that will receive this message. Another problem is that messages may be sent to the IOProc server when they are really meant for some server within the node. For example, if a client does a locate of rs232 relative to `/IO`, the IOProc server will get this request and will need to forward it internally to the RS232 server. In theory the IOProc server should also be able to forward requests to servers on the transputer side of the network, but that is more difficult to achieve.

To see what handlers are required for the IOProc server, look at the header file `fundefs.h`. `IOProc_InitServer` is defined as an external, so this is a real routine that has to be provided. On the other hand, `IOProc_Refine` is `#defined` as Forward, a function which either forwards the request to

another server or sends back an error code indicating inappropriate function for a particular object. These handlers are all defined in the module `server.c`. `IOProc_InitServer()` builds a linked list containing the various entries in the `/IO` directory (that is, the various servers supported in the current configuration). Initially all servers are held in the `WaitingCo` list and the `conode` structure contains the name of the server, so there is no problem about building this list. Once the list has been built it is very easy to support reading directories. `IOProc_Open()` creates a new thread for the stream, using `IOProc_Handlers` for the handler functions. However, these handler functions are `#defined` to be the general-purpose directory reading functions, also used when reading a directory on disc or when reading the `/rs232` directory. `IOProc_TidyServer()`, called when leaving server mode, releases the space taken up by the linked list.

Distributed searches are handled by a private protocol. When `General_Server()` or `General_Stream()` detects a non-GSP message, it will call the handler function for private messages. In most cases this handler returns an invalid function error to the client. However, for the IOProc device, the private protocol message may be a distributed search. If so, the server extracts the name of the server (for example, rs232), tries to find it in the linked list, and if successful it returns success together with the port/thread identifier for the server.

There are the usual handlers to support Create, Locate, and ObjectInfo requests which have to be supported by all servers. Finally there is the Rename request. Renaming is nasty, because it involves extracting both the source and the destination name from the message. A typical Rename would be from `/IO` to `/Cluster/IO`. The reason for supporting this network naming is that the reply to Open, Create and Locate requests must include the full name of the object. For example, if a client opens `/rs232/default` the Open reply contains the name `/Cluster/IO/rs232/default`: this makes it easier to restart the client-server interaction when something goes wrong. The global variable network name contains the current node name (for example, `/Cluster/IO`), and the name conversion routines used in the server leave the object name (for example, `rs232/default`), in variable IOname. Hence constructing the full network name, done in routine `FormOpenReply()`, is fairly easy.

9 Other Servers

In addition to the IOProc device the Server contains a number of different servers, but they all work in much the same way. The header `fundefs.h` defines the handler routines, `server.h` stores them in the table of servers,

`Init()` starts threads for them, and they all start up in `General_Server()`. Obviously the handler routines differ quite considerably between the different servers, as do the local routines invoked by then handlers to do real work, but the underlying organisation is the same.

On some machines such as Sun Workstations there is a single file server called `/files`: on others there are different servers for every disc drive. In addition there is a file server called `/helios`. This server exists to provide the same interface on all the different machines: programs can always access `/helios/lib/cstart.o`, and do not have to be recompiled or reconfigured to access `/c/helios/lib/cstart.o` or `/files/usr/helios/lib/cstart.o`. The filing system is accessed by a number of local routines. For example, when a file server receives an Open request for a file, the handler routine `Drive_Open()` calls the local routine `open_file()`. To port the server onto different hardware it is necessary to provide this `open_file()` routine in the local module, but the Helios interface can remain unchanged.

After a filing system, the most important device is some sort of terminal input/output. On some machines multiple windows are available, and a window server is supported. On others there is only a single window, so a console device is supported instead a window manager may be run on the transputer side to provide multiple shells. All Helios terminal streams must support the ANSI protocols for input and output, and a general ANSI emulator is provided if necessary. If you use this ANSI emulator, it is possible to provide multiple windows on a no-windowing system by using different shadow screens for each: this approach is used in the PC server.

The module `devices.c` contains some additional devices, mostly optional ones. The clock device is used to provide a real-time clock. In fact part of the configuration information sent during bootstrap contains the current time allowing the transputer to maintain its own dock, so the dock device is accessed only occasionally. If it is necessary to run X-windows on the transputer side then the Server must support mouse and raw keyboard devices. There is also code to support RS232, Centronics, Midi, and printer devices: these tend to work in much the same way, with data coming in down a bit of wire and data going out down a bit of wire, so they share a lot of code.

The final device is a GEM VDI server, used to provide graphics output on the PC server and possibly on the ST server at some future stage. This server is slightly different from most in that a private protocol is used between the client and open streams rather than the GSP protocol. However, GSP is used for some of the operations, where it is particularly suitable.

10 The Debugger

The Server contains a simple transputer debugger. This is used mainly when porting the Server to new hardware, to check that the link input/output routines work as they are supposed to, and to debug the Helios nucleus, particularly the kernel hence it is of little use to ordinary users, and there is a `defines.h` option to leave out the debugger completely.

When running in debug mode, the Server maintains a list of symbols associated with particular addresses. For example, the debugger knows that address `0x80000000` is the base of memory, so when you disassemble at that address `MemBase` will be displayed alongside the disassembly. In addition the Helios nucleus contains the names of all the system routines, and there is a `define` command in the debugger to allow you to store all these names in the linked list as well.

The main routine in the debugger, `debug()`, reads data from the keyboard, interprets it as commands, and calls the corresponding routines. Some commands are single key, for example, the `'.'` command is used to display the next frame of memory. Other commands (for example `reset`), have to be typed in full. The debug loop reads in the keys. Alphanumeric keys are stored in a buffer and backspace and delete may be used for editing. Otherwise if the buffer is not empty then it may contain a symbol such as a routine name, which means that the debugger should move to the corresponding address, or it may contain a number which is interpreted as an address, or it may contain a built-in command such as `reset`. If the buffer is empty then the key is interpreted as a single-key command. The default action is to display a frame of the transputer memory as a hex dump.

All the built-in commands are held in a table, together with handler routines. Most of these are fairly simple; for example, the `reset` command just involves doing an `xpreset()` and outputting a message. The `go` command is slightly more complicated: it boots up the transputer and then waits for messages from the transputer, meanwhile allowing the user to send some messages himself. The main use of this command is to check that the system does come up correctly. A related command is `cmp`, which compares the system image loaded in the transputer memory with the system image held on disc. The `clear` command uses one of the options in the bootstrap program `nboot.i` to clear some of the transputer's memory, very useful before a `go` command to make sure that the transputer is in a known state. The `dump` command, used after analysing the transputer, should store the bottom of memory where the transputer holds debug information, and the `info` command can be used to examine it. The `explore` command provides all this in one go. Finally, the `trace` command is used to examine the top of memory where the kernel keeps its trace vector and can store certain debug

information.

The module `dbdecode.c` contains code to disassemble the transputer and to store the various symbols. Disassembling a transputer is fairly straightforward, and details of the instruction set may be found in various Inmos publications.