Helios Standalone Support

PERIHELION SOFTWARE LTD

September 1991
Copyright

This document Copyright © 1991, Perihelion Software Limited. All rights reserved. This document may not, in whole or in part be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent in writing from Perihelion Software Limited, The Maltings, Charlton Road, Shepton Mallet, Somerset BA4 5QE, UK.

This manual is Edition 1.3, September 1991.

Acorn and ARM are trademarks of Acorn Computers Ltd.
Amiga is a registered trademark of Commodore-Amiga, Inc.
Apple is a registered trademark of Apple Computers, Inc.
Atari is a trademark of the Atari Corporation.
Commodore is a registered trademark of Commodore Electronics, Ltd.
Ethernet is a trademark of Xerox Corporation.
Helios is a trademark of Perihelion Software Limited.
IBM is a registered trademark of International Business Machines, Inc.
Inmos, occam, T414, T425 and T800 are trademarks of the Inmos group of companies.
Intel and iPSC are registered trademarks of Intel Corporation.
Macintosh is a trademark of Apple Computers, Inc.
Meiko and Cesius are trademarks of Meiko Limited.
Motorola is a trademark of Motorola, Inc.
MS-DOS is a registered trademark of The Microsoft Corporation.
Parsytec, Paracom and SuperCluster are trademarks of Parsytec GmbH.
POSIX refers to the standard defined by IEEE Standard 1003.1-1988; Posix refers to the library calls based upon this standard.
Transtech is a trademark of Transtech Devices Ltd.
Sun, SunOs and SunView are trademarks of Sun Microsystems.
Telmat and T.Node are trademarks of Telmat Informatique.
Unix is a registered trademark of AT&T.
The X Window System is a trademark of MIT.

Printed in the UK.
PDF generated in AT (Vienna).
Acknowledgements

*Helios Standalone Support* was written by members of the Helios group at Perihelion Software Limited. Helios software is available for multi-processor systems hosted by a wide range of computer types. Information on how to obtain copies of Helios software is available from Distributed Software Limited, The Maltings, Charlton Road, Shepton Mallet, Somerset BA4 5QE, UK. (Telephone: 0749 344345.)
# Contents

1 Introduction 2

2 Host support 3
   2.1 Programmer interface 3
   2.2 Sarun 4
   2.3 Salink 4

3 Virtual Helios environment 5

4 Limited standalone environment 6
   4.1 salib.h 7
   4.2 sysinfo.h 7
   4.3 trace.h 7
   4.4 thread.h 7
   4.5 chanio.h 8

5 Examples 9
   5.1 Sarun 9
   5.2 Exploratory worm 11

6 SALIB summary 12
Chapter 1

Introduction

The Helios standalone runtime system allows the user to run programs in naked transputers without the need for Helios to be loaded. There are several reasons for doing this. The processor may not have enough memory to run Helios (many early transputer cards have only 256k), the program may need to control all the resources of the processor itself (for example, a device interface) or the user may want maximum performance from the program.

The standalone system may be used in two distinct ways. In the first an existing program is run alone for performance reasons and in the second a custom program has been written. These two uses are reflected in the two environments supplied to support standalone programming. The first is a virtual Helios environment in which a program runs as if it were running under a limited version of Helios. Such a program can be tested under Helios and then moved into a standalone environment without recompilation or relinking. The second is a more limited environment in which only a small subset of library routines are available. This is much closer in style to that available under occam or any other standalone language system.
Chapter 2

Host support

The standalone system is designed to be hosted from Helios. Therefore it will only work in a hardware configuration consisting of at least two processors.

2.1 Programmer interface

The support provided to host programs consists of the header <linkio.h> and the module sasup.o. The latter contains three procedures:

- link_open
- link_close
- link_boot

The function link_open (word linkno) opens the given link for raw I/O. The result will be TRUE if it succeeds and FALSE if it fails. Similarly link_close (word linkno) closes the link down and returns it to its original state. The function link_boot (word linkno, char *file) runs the program in the named file in the processor through the given link. The file must have been generated by salink. The result of link_boot is either zero if it succeeded or an error number if it failed. The error numbers are as follows:

- Could not find /helios/lib/nboot.i.
- Could not read bootstrap.
- Could not open program file.
- Could not send bootstrap size.
- Could not send bootstrap.
- Could not send command to bootstrap.
- Could not send program.

The header file linkio.h, in addition to the function templates for the above procedures, contains a set of macros to perform link I/O. These correspond to
CHAPTER 2. HOST SUPPORT

the macros available in chanio.h and are described in section 4.5. The variable _LinkTimeout controls the timeout applied to link transfers. It is initialised to $2*\text{OneSec}$. The failure of link_boot is commonly caused by the target processor awaiting a bootstrap message when it is not in a reset state.

2.2 Sarun

This is a simple program which itself executes a standalone program. It takes two arguments: a link number and a system file generated by salink. Once the program has been run, sarun will transfer any data received on its standard input through the link, and similarly transfer any data read from the link to its standard output. The sarun program will terminate on EOF (or CTRL-D) on its standard input. Therefore a standalone program may be placed in a conventional shell pipeline or even in a CDL script. For example:

```
produce | sarun 2 munge | consume
```

The program munge will run on the processor through link 2. It will get input from produce and its output will go to consume. Care must be taken to ensure that sarun is executed on the correct processor.

2.3 Salink

The program salink is used to prepare a program for standalone execution. It generates a system file which may be passed to link_boot or to sarun. The arguments are as follows:

```
salink [-t4|8] -o dest program
```

The program is a linked Helios program generated in the usual way. The -o flag must be present (it introduces the output file). The -t flag is followed by either 4 or 8, indicating whether the target processor is a T414/425 or a T800.

The program is examined by salink for references to shared library modules. If these are present then a virtual Helios environment is built containing the modules referenced. If the program contains no such references, it can be run on its own and a minimal system is built around it.
Chapter 3

Virtual Helios environment

In this environment certain Helios programs may be run ‘standalone’ without needing to be recompiled or relinked, although most operating system features, such as message passing and file access, are absent. Instead the program is linked with a special standalone version of the Kernel, System library and Posix library. The usual versions of all other libraries are used.

The standalone Kernel lacks all the routines concerned with port manipulation, message passing, event handling, and link configuration and allocation. All other routines such as list handling, semaphores, link I/O and memory allocation function as before. The absent functions, if called, generate a suitable error. The standalone System library is almost entirely non-functional with the exception of \texttt{Malloc} and \texttt{Free}. As in the Kernel, the absent functions will generate an error if called.

The standalone Posix library is also non-functional, with the exception of the routines \texttt{read()}, \texttt{write()}, \texttt{close()} and \texttt{dup2()}. All read and write operations are translated into link transfers. The link used depends on the file descriptor. File descriptors 0 and 1 (\texttt{standard input} and \texttt{standard output}) are mapped onto the processor’s boot link. Descriptors 2 and 3 (\texttt{stderr} and \texttt{stddbg}) are simply duplicates of descriptor 1. Descriptors 4 to 11 are mapped onto the four links in numerical order so that link \( n \) may be read on descriptor \( 2n + 4 \) and written on descriptor \( 2n + 5 \). Descriptors may be closed and the mapping rearranged using \texttt{dup2()} but new streams may not be opened.

All remaining libraries function exactly as before except that any routines which make calls on now absent Kernel, System library or Posix library functions will generate errors. Thus for example, the C library \texttt{fopen()} call will fail. However, the standard C library streams \texttt{stdin}, \texttt{stdout} and \texttt{stderr} will function normally by transferring data through the boot link. The virtual Helios environment may therefore be used to run programs which communicate only through their standard I/O streams.
Chapter 4

Limited standalone environment

The limited standalone environment provides a level of support equivalent to that found in occam and other standalone transputer language systems. Runtime support is limited to the facilities supplied by a small set of headers and a simple library of procedures. These provide the means to use links and channels, create new processes and perform other simple functions. The intended use for this environment is in low-level hardware control or lightweight applications.

Definitions for this standalone environment are found in the headers chanio.h, thread.h, trace.h, salib.h and sysinfo.h. Additionally, some of the routines found in the following normal Helios headers are also supported: stdlib.h, string.h, time.h, math.h, nonansi.h, setjmp.h, queue.h, sem.h, syslib.h.

The code is present in two libraries: salib and samath which come in both T4 and T8 versions.

To build a limited standalone program, compile the program as usual and link it with the libraries. A system image is then generated by salink as before and run via sarun. As an example, consider compiling the program worm.c.

The source is compiled in the usual way, to generate an object file:
```
c -c -o worm.o worm.c
```
This file must then be linked with sastart.o, salib and (optionally) one of the math libraries to produce an executable program:

```
as /helios/lib/sastart.o worm.o -l/helios/lib/salib
-l/helios/lib/samath.t4 -o worm
```

The -l flag indicates that salib and samath.t4 are libraries. Only the modules actually referenced will be linked with the program. Finally, the program must be passed to salink to generate a program which may be executed standalone:

```
salink -o worm.sa worm
```

The following sections describe the standalone-specific routines in SALIB under the headers used to define them.
4.1 salib.h

The header salib.h defines some miscellaneous routines, mostly concerned with the memory management system. memtop may only be called before the first call to malloc. It determines the size of the free memory and returns the address of the first byte beyond the end of store. memtest performs a memory test cycle on the supplied area of memory. malloc_fast and free_fast are used to allocate and free areas of the fast RAM. Since these routines differ from the Helios fast memory allocation routines, Accelerate must be used slightly differently. Where the code under Helios might be:

```c
Carrier *carrier;

carrier = AllocFast(size,&MyTask->MemPool);
Accelerate(carrier,fn,sizeof(int),x);
Free(carrier);
```

The code in the standalone environment is:

```c
Carrier carrier;

carrier.Addr = malloc_fast(size);
carrier.Size = size;
Accelerate(&carrier,fn,sizeof(int),x);
free_fast(carrier.Addr);
```

4.2 sysinfo.h

The header sysinfo.h defines the SysInfo structure, which is initialised to contain some useful values. These values are the base address of the free memory in the processor, the address of the program’s module table and the identity of the processor’s boot link. A pointer to the trace vector is also kept here but it is only initialised if _TraceInit is called. A macro called SYSINFO is supplied to return the address of the SysInfo structure.

4.3 trace.h

The standalone library contains copies of the _Trace, _Mark and _Halt routines defined in the Helios Kernel. These all place entries into a trace vector at the top of the memory, which may be examined with the debugger. The trace vector must be initialised with _TraceInit before entries may be made.

4.4 thread.h

Most programs can use the Helios-compatible Fork procedure to create new processes. However, the procedure thread_create, defined in this header, may be used where a more primitive form is required. Fork will normally allocate
the stack of a new process using `malloc`. If you do not want your program to initialise the memory system then you may use `thread_create`, which passes a pointer to the top of the memory to be used as the stack. For example,

```c
static char stack[2000];

thread_create( stack+2000, 1, fn, sizeof(x), x );
```

runs `fn` as a parallel process at low priority, using the array `stack` as its stack. No vector stack is created and the module table is passed directly to the function. Therefore, any C program must be compiled with `#pragma -s1 -f0` at the top to switch off stack checking and the vector stack. The `thread_stop` procedure simply halts the calling process.

### 4.5 `chanio.h`

The `chanio.h` header defines a number of macros to be used for channel and link I/O. These macros are named according to the following convention: `medium_direction_item` where `medium` is either `link` or `chan`, `direction` is either `in` or `out`, and `item` is `byte`, `word`, `struct` or `data`. The first argument is always either a pointer to the channel or the number (0–3) of the link to be used for transfer. The second argument is either the data item to be transferred, or a pointer to it. Where `item` is `data` a third argument defines the size of the data to be transferred. For example, the following code transmits a word, followed by a string, and receives a data structure back from a link:

```c
char *name;
struct Answer answer;

link_out_word(link,123);
link_out_byte(link,strlen(name));
link_out_data(link,name,strlen(name));

link_in_struct(link,answer);
```

The function `alt` performs an `occam`-style alternate on an array of channel pointers. The first argument is a time interval in ticks of the current priority clock. If it is zero, a non-timer alternate is performed. The second argument is the size of the channel array. The third argument is a pointer to an array of channel pointers. The result of the `alt` function is either the index in the array of the channel which was selected, or -1 if a timeout occurred. Unlike the `occam` construct, if a channel is selected, the input is not performed by the `alt` routine, and must be performed by the subsequent code. The procedure `boot` attempts to boot a copy of the program into the processor through the link given.
Chapter 5

Examples

This chapter illustrates, with some examples, how the standalone system may be used.

5.1 Sarun

The sarun program is used to load and execute a program on a naked processor. It shows how the host support routines are used.

```c
/* SARUN - 20/8/89 */
/* Program to load and run a stand alone program */
/* To compile: c sarun.c sasup.o -o sarun */

#include <stdio.h>
#include <stdarg.h>
#include <linkio.h>
#include <stdlib.h>
#include <sem.h>
#include <nonansi.h>

Semaphore sync; /* Termination synchronisation */
bool finished = FALSE; /* set True on EOF */

static int error(char *f,...)
{
    va_list a;
    va_start(a,f);
    vfprintf(stderr,f,a);
   putc( '
', stderr );
exit(1);
}

/* Input process */
/* Reads characters from STDIN and passes them through link */
void input(word link)
{
    word c;
```
do {
    c = getchar();
    if( c == EOF ) break;
    link_out_byte(link,c);
} while( !finished );
finished = TRUE;
Signal(&sync);
}

void output(word link)
{
    word c = 0;
    do {
        word e = link_in_byte(link,c);
        if( e < 0 ) continue; /* timeout, just loop */
        putchar(c);
    } while( !finished );
    finished = TRUE;
    Signal(&sync);
}

int main(int argc, char **argv)
{
    int e;
    int link;

    if( argc < 3 ) error("usage: sarun link bootfile");
    link = atoi(argv[1]);

    /* open the link for raw I/O */
    if !link_open(link) ) error("failed to open link %d",link);
    /* boot the program through it */
    if( (e=link_boot(link,argv[2]))!=0 )
        error("link_boot error %d",e);
    /* now spool stdin into the link and */
    /* anything from the link to stdout */
    InitSemaphore(&sync,0);
    Fork(2000,input,4,link);
    Fork(2000,output,4,link);
    Wait(&sync);
    Wait(&sync);
    if( !link_close(link) ) error("failed to close link %d",link);
}
5.2 Exploratory worm

This is a recoding into C of an occam worm. It has two parts: the program explore which runs under Helios and the worm, which is run on naked processors. The program as it stands is intended to generate a resource map for an unknown processor network. It assumes that Helios has been booted into the first processor of the network and explore has been run there. It checks each link, identifying booted processors and passing the worm through unused links. Then a text resource map, which may be captured and compiled, is produced as output. Another use of this worm is to flood-fill a processor network with copies of a particular program. The worm may then generate a load balancing task farm. The Helios standalone support package contains the source of the worm. It is in the examples directory, under the directory sa. (For example: /helios/users/guest/examples/sa.)
Chapter 6

SALIB summary

This chapter lists the functions supplied by SALIB under the headers used to define them.

```c
#include <stdlib.h>
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void exit(int status);

#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
size_t strlen(const char *s);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(char *s1, const char *s2);
void *memset(void *s, int c, size_t n);
```

```c
#include <time.h>
extern clock_t clock(void);
```

```c
#include <math.h>
extern double acos(double x);
extern doubleasin(double x);
extern doubleatan(double x);
extern doubleatan2(double x, double y);
extern doublecos(double x);
extern doublesin(double x);
extern doubletan(double x);
```
extern double cosh(double x);
extern double sinh(double x);
extern double tanh(double x);
extern double exp(double x);
extern double frexp(double value, int *exp);
extern double ldexp(double x, int exp);
extern double log(double x);
extern double log10(double x);
extern double modf(double value, double *iptr);
extern double pow(double x, double y);
extern double sqrt(double x);
extern double ceil(double x);
extern double fabs(double x);
extern double floor(double d);
extern double fmod(double x, double y);

<stdio.h>
void *NewProcess(WORD stacksize, VoidFnPtr function, WORD argsize);
void RunProcess(void *process);
void ZapProcess(void *process);
WORD Fork(WORD stacksize, VoidFnPtr function, WORD argsize, ...);
void Accelerate(Carrier *fastram, VoidFnPtr function, WORD argsize, ...);
void AccelerateCode(VoidFnPtr function);
void IOdebug(const char *fmt, ...);
void IOputs(char *s);
void IOputc(char c);

<setjmp.h>
extern int setjmp(jmp_buf env);
extern void longjmp(jmp_buf env, int val);

<queue.h>
PUBLIC void InitList(List *);
PUBLIC void PreInsert(Node *, Node *);
PUBLIC void PostInsert(Node *, Node *);
PUBLIC Node *Remove(Node *);
PUBLIC void AddHead(List *, Node *);
PUBLIC void AddTail(List *, Node *);
PUBLIC Node *RemHead(List *);
PUBLIC Node *RemTail(List *);
PUBLIC word WalkList(List *, WordFnPtr, ...);
PUBLIC Node *SearchList(List *, WordFnPtr, ...);

<sem.h>
PUBLIC void InitSemaphore(Semaphore *, word);
PUBLIC void Wait(Semaphore *);
PUBLIC void Signal(Semaphore *);
PUBLIC void SignalStop(Semaphore *);
PUBLIC word TestSemaphore(Semaphore *);

<syslib.h>
void Delay(word usec);

<salib.h>
extern void *memtop(void);
extern int memtest(word *base, int size);
extern void boot(int link);
external void freestop(void *addr);

<stdio.h>
_SYSINFO

<trace.h>
external void _TraceInit(void);
external void _Mark(void);
external void _Trace(int x,...);
external void _Halt(void);

<thread.h>
external void thread_create(void *stack, word pri, VoidFnPtr fn, word nargs,...);

<chanio.h>
chan_out_byte(c,b)
chan_out_word(c,w)
chan_out_data(c,d,s)
chan_out_struct(c,d)
chan_in_byte(c,b)
chan_in_word(c,w)
chan_in_data(c,d,s)
chan_in_struct(c,d)
link_out_byte(l,b)
link_out_word(l,w)
link_out_data(l,d,s)
link_out_struct(l,d)
link_in_byte(l,b)
link_in_word(l,w)
link_in_data(l,d,s)
link_in_struct(l,d)
external int alt(int timeout, int nchans, Channel **chans);
Index

chanio.h - standalone header, 8
Environment
  limited standalone, 6
  virtual Helios - standalone, 5
Exploratory worm - standalone, 11
Host
  support, 3
Programmer interface
  standalone, 3
SALIB summary - standalone, 12
salib.h - standalone, 7
salink program - standalone, 4
sarun, 9
sarun program - standalone, 4
Standalone
  chanio.h, 8
    exploratory worm, 11
    host support, 3
    limited standalone environment, 6
    programmer interface, 3
    SALIB summary, 12
  salib.h, 7
  salink program, 4
  sarun, 9
  sarun program, 4
  support, 2
sysinfo.h, 7
thread.h, 7
trace.h, 7
  virtual Helios environment, 5
Support
  standalone, 2
  host support, 3
sysinfo.h (standalone), 7

thread.h, 7