Helios System Software Series

Software Documentation

Author :
Hans-Joachim Ermen

The Helios Application Guide

(including Helios 1.15 Release Notes)

January 1990

Contents

Introduction

During the last two years, the Helios Operating System has become a widely accepted standard development and application interface in parallel computer environments, based on the Transputer. It includes a set of versatile and powerful tools for the development and execution of parallel applications. With the latest release on Helios, called "Helios 1.15", which is directly based on the release 1.1, many improvements and enhancements in functionality, basic reliability and performance have been added to the system.

This guide is intended to give the user a practical guideline to make better use of Helios and to demonstrate some standard methods and procedures, which can be used during daily work with the system. The features added in Helios 1.15 are particularly covered in depth. Points of deeper discussion are topics, which are only touched upon in the original documentation with a consequent lack of a practical point of view or which tend to produce misunderstanding. Beyond this, the manual may be helpful to everyone who wants to come into closer contact with this unique operating system. Many questions, asked by developers and users during the last year have been taken into consideration.

It is assumed that the reader is familiar with the basic concepts of Helios as described in the original documentation ("The Helios Operating System"). This guide is neither a Helios introduction, nor should it replace the original documentation as supplied by Perihelion Software Ltd!

1. Helios Release 1.15

Topics:

- The version history
- What are the actual development aims ?
- Modifications and enhancements

Helios 1.15 is directly based on the release 1.1 in the single user version (Helios /s). An intermediate version - named Helios 1.1A - was created by Perihelion in close co-operation with Parsytec and is fully supported by Perihelion and Parsytec. Some additional features were added by Parsytec and incorporated into Helios release 1.15. Nevertheless the basic functionality of Helios 1.15 and of Helios 1.1A, is nearly the same as Helios 1.1. A subset of features which were basically implemented in the Helios network-version (Helios /n) are left out, but will be included into the next official release.

Some of the most important enhancements take place in the areas of networking and Task Force Management to support very large Transputer networks with much more than 64 processors, active under the control of one or an arbitrary number of users. It is now possible to support a wide range of different topologies and to run Task Forces with a nearly indefinite number of Task components which are executed in parallel. The Helios operating system kernel routines - called Nucleus - has also been improved to optimize the system's long term behaviour and give a maximum of robustness in spite of a heavy usage of the available processor resources. In addition to this improvement of "system imminent" features, the set of standard utilities was also touched upon. Some other user commands were added or extended, including the Norcroft C-Compiler, which is now available in the version 1.17.

Most of the modifications made to the system were heavily tested by Perihelion and Parsytec in different environments on a wide range of Parsytec hardware - from a single processor machine to a SuperCluster with 64 processors and electronic configuration facilities up to a fixed- topology machine with 144 processors - but should also work on most of the other vendors' machines without major problems.

1.1 Installation Notes

Topics:

- Move from Helios 1.1 to Helios 1.15
- Differences in the file arrangement

Helios version 1.15 is delivered in the same format as the original release and can be directly copied onto an existing Helios directory tree. Please note, that the PC-disk contains only an additional set of files as a supplement, which have to be moved together with the Helios 1.1 release files, if you are creating a new system from scratch (1).

On a PC-hosted system, the following command can be used to install Helios 1.15 from MS-DOS level .(For this example it is assumed, that a Helios 1.1 directory tree is residing on a fixed disk partition c: under \helios. The Helios 1.15 supplement disk may be inserted into disk drive a:.)

```
>     xcopy   a:   c:\helios   /s     <CR>
```

The arrangement of the different files under Helios 1.15 differs from Helios version 1.1 in the way, that the device driver for reset & analyze is now resident in the /helios/lib - directory. The user has to copy the desired driver from his etc - directory into the lib - directory. This can be done for example for the Parsytec device driver "*pa_ra.d*" under MS-DOS in the following way:

```
>     copy   c:\helios\etc\pa_ra.d    c:\helios\lib    <CR>
```

(1) : If you are using a SUN as your host system, you get a complete Helios version, which has to be installed as described in the installation guide for Helios version 1.1 for the SUN.

In this case, the standard boot program *rboot* has to be replaced by the Parsytec specific binary object "*pa_rboot*":

```
>       copy   c:\helios\lib\pa_rboot     c:\helios\lib\rboot     <CR>
```

After finishing the installation of the Helios 1.15 files, the system can be booted in the same way as before. Note that the *Network Server* and the *Task Force Manager* have now a serial number and an initialization message is printed after successful start of each of them [2].

(2) : A note for users running X-Windows on a GDS-board: To avoid problems, it is actually recommended to make no use of the *steal*-utility!

2. Resource Management

A set of processors, building a Helios network, is described in a "resource map". Each processor - forming a node in the network topology - is given an unique name and all link interconnections which are used, have to be specified. In addition to this, a set of attributes can be used to describe a certain processor, its resources and usage in a more detailed way. A pre-compiled binary object, which is derived from the text-file of a resource map by the "resource map generator" (*rmgen*), is inspected during the system's boot phase and gives the *Network Server* (NS) all information needed to boot a complete network subsequently.

Because the resource map generator performs only very basic error checks, the user should take great care to write a properly defined resource map!

The declaration of processors and link connections does not have to describe essentially the complete existing physical topology. It is also possible to subdivide a pool of processors into different separate Helios networks, each controlled by its own *Network Server* without affecting the others. As described in chapter 5.5, it is in particular possible, to establish communication links between these different subnets and to share subnets between different users.

The resource map generator (*rmgen*) is sensitive in interpreting a command line. To avoid problems, the following calling sequence is preferable:

```
%      rmgen  -o  file.map  file.rm   <CR>
```

2.1 Resource Map Structure Details

Topics:

- The general layout policies
- Use of hierarchical subnets
- The "Master Node"
- How should processors be arranged within a resource map ?

Helios organizes a collection of processors in a resource map in the way, that it builds logical sub-networks, which are controlled by a centralized instance called *Network Server* (NS). The usual way to define a Helios network is, to put the complete set of processors into one subnet-structure. This approach is called a **"non- hierarchical network"**.

The *Network Server* is also able to handle hierarchical subnet-structures. In this case, different groups of processors are put into different subnets, usually directly bound to a certain network topology; building a tree for example. Because this adds system administration overhead to the network controlling routines and makes the usage of Helios naming mechanisms much more complicated and inefficient, the user should try to avoid hierarchical subnets as far as possible! The only justification for declaring hierarchical subnet-structures occurs when Transputer hardware from different vendors is used to build one physical Helios network. In this case, it is essential to be able to specify different reset & analyze schemes for the different underlying Transputer hardware. Under all other conditions, a non-hierarchical subnet description is preferred.

One node in each subnet has to be declared as a "Master Node", responsible for keeping control over booting the rest of a network. This is done by the *Network Server* running on this node and some other binary objects, which are passed through the network by the *Network Server*, following a certain boot-path. In addition to this, the Master Node processor also has to be noted as a "controller"

(CONTROL-attribute) directly after the beginning of a subnet-specification. In a non-hierarchical network, the Master Node is always identical with the first processor in the subnetwork, known as the *"root processor"*. Therefore, working with non-hierarchical networks seems to make the definition of a Master Node and a controller processor obsolete, but nevertheless, they have to be specified!

In conjunction with the Master Node attribute, the desired reset & analyze device driver is specified as an argument. If Parsytec hardware is used, the device driver *"pa_ra.d"* has to be placed as an argument.

A typical resource map, describing a network with one processor may look similar to the following: (It is recommended, that the root processor is always connected with the host's link interface by using link 0!)

```
--  This is an example for a resource map which describes
--  a network consisting of one processor.

subnet  /Cluster  {
        CONTROL  Rst_Anl [/Cluster/00];

        terminal 00 { ~IO, , , ;        HELIOS;
                                        Mnode Rst_Anl [pa_ra.d];
                                        ptype T800;
                                        }
        terminal IO { ;  IO; }
}
```

(Listing 1)

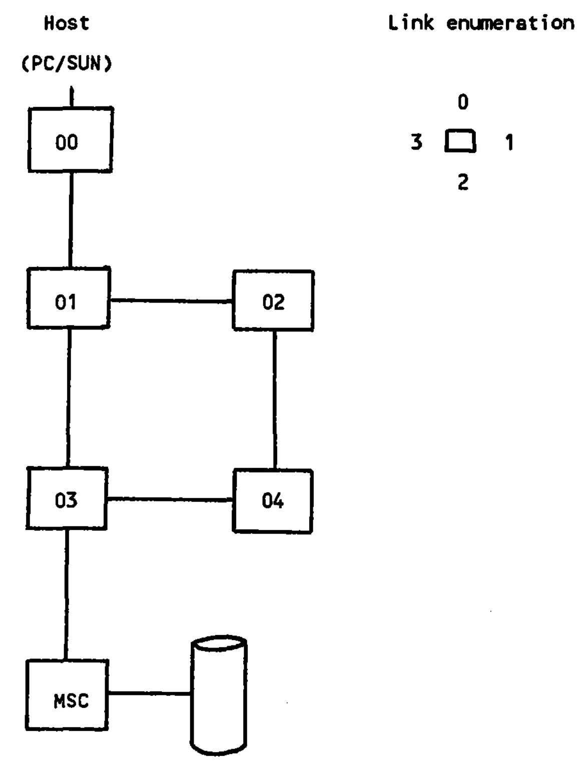
Note, that you can insert comments into your resource map script files by using the OCCAM-like preamble "--" as you can see it in the first two lines of Listing 1.

A processor grid, consisting of four Transputers may be specified in the following way: (Another node - the MSC mass storage node - is connected to processor "03".)

```
-- This is an example resource map

subnet /Cluster {
      CONTROL  Rst_Anl [/Cluster/00];

      terminal 00 { ~IO, ,~01 , ;      SYSTEM;
                                       Mnode Rst_Anl [pa_ra.d];
                                       ptype T800;
                                       }
      terminal 01 { ~00,~02,~03,;      HELIOS;
                                       memory #100000;
                                       ptype T800;
                                       }
      terminal 02 { ,,~04,~01 ;        HELIOS;
                                       memory #100000;
                                       ptype T800;
                                       }
      terminal 03 { ~01,~04,~MSC,;     HELIOS;
                                       memory #100000;
                                       ptype T800;
                                       }
      terminal 04 { ~02,,,~03 ;        HELIOS;
                                       memory #100000;
                                       ptype T800;
                                       }
      terminal MSC { ~03,,, ;          SYSTEM;
                                       memory #400000;
                                       ptype T800;
                                       }

      terminal IO { ;  IO; }
}
```

(Listing 2)

This resource map describes a physical topology like this:



(Figure 1)

The *Network Server* tries to boot the processors declared in the given resource map by using a "breadth-first" algorithm. (A more detailed description of the boot mechanism is given in chapter 5.2.) Writing a resource map, the user has to take into account, that the order of processors within the resource map influences directly the initial boot sequence. It causes especially major problems, if a processor is specified in the resource map and picked up by the *Network Server*, before it has been succesfully booted by one of the processors declared before. This is the main reason for problems users usually have while booting certain network topologies, especially different kinds of meshed structures!

A practical example: It is not possible to boot the network shown above successfully, if the specification of processor "04" follows in the resource map as the second declaration directly after processor "00" has been declared, because processor "04" has not been booted by any other active node yet!

If a network consists of only one "isolated" processor, the installation of a *Network Server* and a *Task Force Manager* is not recommended, because there are no distributed resources to be managed. To prevent the system from installing them, just put the associated lines in the "initrc" file into comments. The initrc - file can be found in the etc - directory. (3)

(3) : See chapter 5 for more details about booting of multiple processor networks..

2.2 Naming of Network Elements

Each individual subnet can be given a name free of the user's choice. As a convention, most of the example resource maps use "Cluster" as the name of the subnet, but there is no restriction to call a subnet maybe "Net" or "Hippopotamus".

In the same way, each processor has to be given an unique name by the user. To keep the subnet structures clear, a well established method is to enumerate all processors which can be defined as pure processing nodes with digits, whereas processors, which have to perform special tasks - like generating graphics or accessing mass storage devices - are named by using letters.

A special note for the users of Parsytec MultiCluster or SuperCluster machines: The utility program *"hconfig"*, which examines a binary resource map and sends a request to the *Network Configuration Manager* (NCM) to allocate processors and configure them, makes some assumptions on the processor naming scheme: It is assumed, that all processors which are part of a dynamic partitioned and configured network, are enumerated numerically beginning with "01". Therefore you have to take great care by choosing processor names to avoid conflicts with "external" nodes and MultiCluster- or SuperCluster- processors (4).

By default, the root-processor's name is set to "00". The pseudo I/O-node, which allows access to the host's external devices is usually called "IO". Please keep in mind, that you have to change also your *host.con* file, if you want to give your root-processor or I/O-node another name than "00" or "IO".

(4) : A more detailed description of partitioning and configuring a MuliCluster or SuperCluster machine can be found in the associated system software documentation.

Imagine a network consisting of a root node, 16 "general processing nodes" and a graphics node running the X-Window environment. The following naming scheme may be used for best clarity:

00	=	root processor
01	=	first processor
02	=	second procesor
...		...
16	=	
GDS	=	a graphics node (Parsytec GDS board)
IO	=	pseudo I/O-node

(Figure 2)

Taking the example resource map for the one- processor network from above and changing the root's name to "MY" and the name of the I/O-node to "PC" results in the following script. The complete subnet shall have the name "PCNet" and the Parsytec reset & analyze device driver is used:

```
subnet  /PCNet  {
        CONTROL  Rst_Anl [/PCNet/MY];

        terminal MY { ~PC, , , ; HELIOS;
                                 Mnode Rst_Anl [pa_ra.d];
                                 ptype T800;
                                 }
        terminal PC { ;  IO; }
        }
```

(Listing 3)

In addition to these changes, the lines

```
root_processor    =    /MY
io_processor      =    /PC
```

have to be added to your *host.con* file.

If you have modified your *host.con* file, you have to exit from Helios and restart the server from your DOS or UNIX shell to make the system notice the modifications. (A restart using the hot-key combination SHIFT+CTRL+F10 under MS-DOS does not open the *host.con* file for re-initialization!)

2.3 Attributes

Various attributes can be added to a certain processor description within a resource map to determine the usage of this resource in a more detailed way. The following subset of attributes should be used whenever a network is specified; especially if Task Force management facilities are required:

i) Processor usage : HELIOS / SYSTEM

Declaring a Transputer as a "HELIOS"-node allows the *Task Force Manager* as a global network instance, to place components of a Task Force on this processor. If this is to be avoided, a processor has to be noted as a "SYSTEM"-node in the resource map! In this case, the mapping routines of the *Task Force Manager*, which have to decide where to execute different components of a Task Force, ignore this node completely. If the *Task Force Manager* is not involved, it makes no difference, whether a processor is declared as SYSTEM or HELIOS. For example, it is still possible, to place explicitly Tasks using "remote" on a SYSTEM-node!

It is especially sensible to declare a processor as a SYSTEM node under the following **two conditions**:

1) If a certain processor is not an "ordinary" processor in the way that it offers perhaps access to external hardware like mass storage devices or bit map graphics. To avoid problems with the server programs running on these nodes, influenced by user Task Forces, they should always be given the SYSTEM-attribute.

2) If you are using a large Helios network controlled by a root processor which is equipped with limited memory, its declaration as a SYSTEM-node guarantees, that memory is not used up by Task Force components. Because the *Task Force Manager* allocates various chunks of memory dynamically, if a large number of Task Forces have to be controlled, it is a good approach to keep the root processor in general free from all other network-wide activities like executing components of a Task Force by using the SYSTEM-attribute, to avoid memory bottlenecks.

Other processors can also be declared as SYSTEM nodes, if some more resources are required, which should not be influenced by Task Forces running in the background. For example, if two dedicated processors within a 32-processor system are specified as SYSTEM-nodes, they can be used for editing, compiling etc. without demands by Task Forces, which are running "in the background" and making use of the remaining 30 processors, which are declared as HELIOS-nodes.

ii) Memory attribute : memory

For each processor, the size of external memory can be specified by using the "memory" attribute. This is especially recommended, if a node has a memory size different from 1 MByte.

The memory attribute is also used by the *Task Force Manager* during the initialization phase of a Task Force to determine the amount of memory available for task components on a certain processor. In combination with the memory attribute as supplied by component declarations in a CDL-script (5), the *Task Force Manager* is able to "look ahead" and calculate the number of components of a Task Force, which can be placed on a single processor.

(5) : *CDL* stands for "Component Distribution Language", which is a script language used for describing parallel applications. A more detailed discussion follows in chapter 3!

If a node is equipped with 4 MBytes for example, the following line has to be added to the processor description in the resource map:

```
memory              #400000;
```

iii) Processor type : ptype

Using Transputer hardware in conjunction with Helios allows the user to choose between the T414 and the the T800 processors. This information is taken by the *Task Force Manager* to detect, which processors offer floating point support and which are only able to emulate floating point operations. This influences directly the mapping of components, if some of them require floating point support.

If a node is equipped with a T800, it should be specified in the resource map in the following way:

```
ptype        T800;
```

iv) External Links : ext[]

If it is desired to combine a Helios network with other independent Helios networks or with processors running an OCCAM application, the interconnecting links have to be specified. Because the *Network Server* is not able to get informations about alien Helios or OCCAM networks during the boot phase, these links have to be declared as "external".

Remember the example shown in figure 1: If we want to make use of an external link, connected to processor "02" (link 1), we have to change the processor declaration in the following way to get an "external" link. (Please note, that the resource map does not give any information about the processor, which is connected to the external link.)

```
-- Example processor description with link 1 declared
-- as an external link. In this case, we are using a
-- T414 Transputer

     terminal 02 { ,ext[0],~04,~01 ; HELIOS;
                                      memory #100000;
                                      ptype T414;
                                      }
```

(Listing 4)

The index within the external declaration is used to distinguish multiple external links within one subnet. If we want to add another external link, we have to increment the index by using ext[1], ext[2] and so on.

3. Parallel Programming

The parallel programming model of Helios is based on a system service called the *"Task Force Manager"*, which is responsible for distribution and execution-control of parallel applications, named *Task Forces*. A script language, called *Component Distribution Language* (CDL), which is used to express parallelism within a Task Force, builds the user interface for application specification. The main difference in comparison with all other parallel programming models for Transputers is, that each *component* of a Task Force under Helios is written in a "standard" sequential programming language like C, FORTRAN, Pascal or Modula-2. The management of parallel applications is done here at an operating system level. There is no need to add language extensions or library functions to express parallelism! This makes it much more easier to port larger software packages onto a parallel computer running under Helios (6).

The examples given with this documentation are written in C and FORTRAN-77. These two Helios compilers are actually the two programming tools, which should be preferred, because of their full implementation, language specification conformance following the particular ANSI-standards and - last but not least - their high reliability.

(6) : To come into first contact with parallel applications under Helios, the "Parallel Programming Tutorial", published by Perihelion Software Ltd, is worthwile reading.

3.1 The Programming Model

Topics:

- How to outline a parallel application under Helios

The process of parallel program development under Helios can be divided into three separate steps:

- At first, the user has to think about possibilities to split up an application or an algorithm to build a number of **independant tasks**, which can be executed in parallel. These tasks are written, compiled and tested separately by using standard sequential programming languages.

- The second step is, to define **communication channels** between these different tasks. The communication mechanism used, is based on a model of "pairs of associated streams", directly following the POSIX conformance specification for stream-I/O. Different tasks are reading and writing from different POSIX-streams, which are connected transparently by the *Task Force Manager* by using pipe-servers.

- There is no information contained within an individual task component which gives any hint about the "other side" of such a communication channel. This gap has to be filled by the user by writing a "CDL-script". During this third step of development, the **whole Task Force** as a parallel application - built of several *task components* - has to be **described.** Each component can be specified in various ways and it is especially possible, to define the set of corresponding streams. With a CDL-script, different independant task components are joined together to build a homogeneous unity - a parallel application!

3.2 Usage of POSIX Streams

Topics:

- Why should POSIX streams be preferred?
- The POSIX stream enumeration schema
- Error handling using POSIX calls

As mentioned before, the communication mechanism for Task Forces is based on "Streams". In the first instance, it makes no difference, whether C-streams, POSIX-streams or streams at the Helios system-call-level are used, but there are some good reasons to prefer the usage of POSIX-streams:

- C-streams offer comfortable routines for data transfer, but add a lot of overhead to an application. The behaviour of C-streams is a little bit unforseen because by default, internal buffering mechanisms are used. As a consequence, working with C-streams results in the lowest performance in comparison to all other Helios communication machanisms and should be avoided!

- The Helios-streams build a layer directly based on the basic Helios message passing surface and offer better data transfer rates than using C-streams. On the other hand, the corresponding routines for handling Helios-streams are unique to this operating system, which makes it much more difficult to port existing software from another machine.

POSIX-streams give a good compromise between these two alternatives: On one hand they are familiar to all UNIX-users - well defined since UNIX version 7 - on the other hand, working with POSIX-streams under Helios offers nearly the same data transfer rates than using Helios-streams directly. In conjunction with the POSIX compatible calls *read()* and *write()*, nearly all communication models for typical Task Forces can be realized. Because of the fact, that not only the

Helios C-compiler, but also the FORTRAN, Pascal and Modula-2 compilers have a POSIX-compatible library interface. Consequently it is very easy to write portable applications and mixed-language Task Forces based on the POSIX-standard. These are good reasons to concentrate on the POSIX communication mechanisms, when writing a Task Force.

The POSIX-streams are enumerated beginning with "0" and have the following specific meaning (7):

Stream		Meaning	Direction
0	=	stdin	<--
1	=	stdout	-->
2	=	stderr	-->
(3	=	stddbg)	
4	=	auxiliary 1 - input	<--
5	=	auxiliary 1 - output	-->
6	=	auxiliary 2 - input	<--
7	=	auxiliary 2 - output	-->
(... and so on ...)			

(Figure 3)

(7) : The *stddbg*-stream should not be used within a Task Force. If debug-messages have to be displayed, the use of the *stderr*-stream or the *IOdebug()* call should be preferred.

To keep the standard streams "free" for I/O-usage and error handling, it is preferable to use the auxiliary streams for communication between different task components. This has to be done explicitely by using the *stream*-attribute within a component declaration and performing *read()* and *write()* calls to streams, beginning with the auxiliary1-input (8). The convention for the usage of auxiliary streams is to read from streams with an even number (4,6,...) and to write to streams, which have an odd number (5,7,...).

The number of streams to a Helios I/O-server is limited and depends on the memory available on the host machine. The user should take care - especially in a PC-hosted environment - that his application Task Forces do not open too many streams (about 50 or more) to the I/O-server.

A common sloppiness, directly concerned with the usage of the POSIX-calls *read()* and *write()*, is the lack of a proper return-code checking within the application. It is the responsibility of the user, to inspect the return value of the POSIX-calls to find out, how many bytes actually have been transferred! Under many circumstances, this is one of the main reasons for a "strange" behaviour of a Task Force running under Helios.

The following little procedures may give an idea, how POSIX stream-I/O can be performed, based on a proper return-code handling (9):

(8) : As shown below in greater detail, there are some variants of Task Force declarations, which do not allow an absolutely free usage of streams.

(9) : The inspection of the global variable *errno* can give more hints to identify the reason for a read or write failure.

```
#include <helios.h>
#include <posix.h>

/*************************************************
 * Read operation with error handling
 *************************************************/
int
full_read  (int stream, char *buf, int amount)
{
     int result;

     while (amount > 0)
     {
         result = read (stream, buf, amount);
         if (result <= 0)
              return -1;
         amount -= result;
         buf = &(buf[result]);
     }
     return 0;
}

/*************************************************
 * Write operation with error handling
 *************************************************/
int
full_write  (int stream, char *buf, int amount)
{
     int result;

     while (amount > 0)
     {
         result = write (stream, buf, amount);
         if (result <= 0)
              return -1;
         amount -= result;
         buf = &(buf[result]);
     }
     return 0;
}
```

(Listing 5)

Using a *full_read()* or the similar *full_write()* procedure guarantees, that the request is always fulfilled with the only termination condition arising, if a fatal error like a broken stream connection occurs. In this case, "-1" is returned to the calling routine. In all other cases, especially if only a part of a set of data could be transferred, a retry is made to get or send the remaining bytes. For example, if 20 KBytes are desired to be read from another task and only 16 KBytes are received with one data transfer, a second try is made to get the remaining 4 KBytes.

In general the user should make sure, that task components within a Task Force are terminated correctly, if transfer operations fail because of fatal error conditions.

3.3 Writing a CDL script

Topics:

- Differences between Helios 1.1 and Helios 1.15
- Component declarations (An example)
- Useful attributes within a CDL-script
- Automatic vs. static mapping of streams

The CDL-compiler under Helios 1.15 offers basically the same functionality as the CDL-compiler under Helios version 1.1. There are some restrictions concerning more "esoteric" features within the *Component Distribution Language*, which are no longer supported under this version. These are especially the *life-*, *time-* and the *priority-* attribute. Some other limitations, existing under Helios 1.1, like the number of streams within a Task Force are now removed. A lot of well-known bugs within the CDL-compiler are also fixed now and the listing output has been be improved.

Interpreting a command line by the CDL-compiler is still quite sensitive. To generate a precompiled CDL-object, the following sequence of parameters within a command line is preferred.

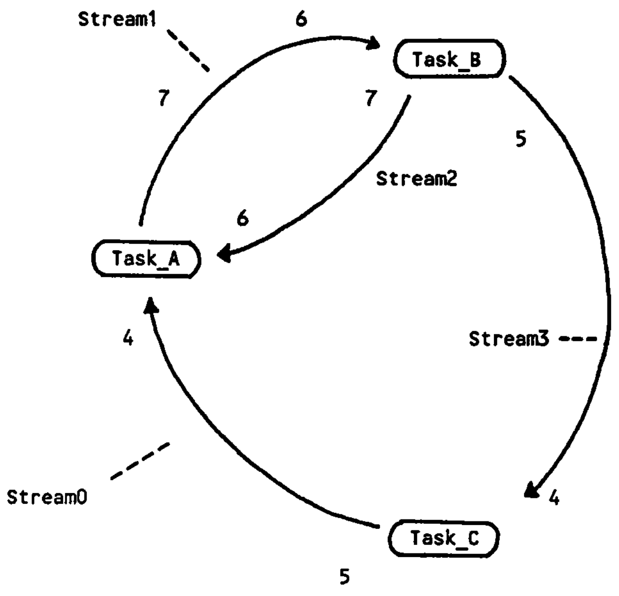
```
%      cdl   -o   target     source.cdl     <CR>
```

To write a CDL-script, the user has first to decide, whether to force the CDL-compiler to manage stream connections for different task components transparently or whether he wants to define pairs of associated streams by himself, using component declarations. If the general parallel operator "^^" is taken, which gives no information about the communication structure between different Task Force components, the user has always to specify explicitly the stream connections within each component declaration. If other parallel operators, like unidirectional pipelines ("|") or farming constructs ("|||") are chosen, the CDL-compiler assumes, that certain combinations of fixed pairs of streams are used to connect different task components. In this case, the user has to take care, that the *read()* and *write()* POSIX-calls within his application programs match these fixed pairs of streams.

To make the process of stream allocation clearer, we show **two examples**: The first one is based on the general parallel operator "^^" and makes use of explicitely allocated streams. The second one works with bidirectional pipe-operators ("<>") and uses a set of streams, allocated by the CDL-compiler. In this case, the most important **attributes**, which can be used in a component declaration, are explained.

1) The first example describes a "dummy" Task Force, consisting of three components, which are specified in separate component declarations. To be completely independant from CDL-restrictions, we decide to declare the communication channels on our own by using the *stream*-attribute within the component declarations. This conditions the usage of the general parallel operator "^^" to describe the relationship between the different independent components:

1) The parallel application:



(Figure 4)

2) The resulting CDL-script:

```
#!/helios/bin/cdl

# These are the component declarations:

component Task_A {
      puid        /Cluster/01;
      code        master;
      memory      100000;
      streams     ,,,,<|Stream0,,<|Stream2,>|Stream1;
}
component Task_B {
      code        slave;
      memory      100000;
      streams     ,,,,,>|Stream3,<|Stream1,>|Stream2;
}
component Task_C {
      memory      100000;
      streams     ,,,,<|Stream3,>|Stream0;
}

# This line describes the parallel application in general

Task_A  ^^  Task_B  ^^ Task_C
```

(Listing 6)

This example Task Force consists of three components, which are connected via three communication channels, modelled as pairs of POSIX-streams.

The *streams* attribute specifies a list of POSIX-compatible streams, beginning with *stdin*, which is equivalent to the POSIX-stream number zero (figure3). The identifiers "Stream0", "Stream1", "Stream2" and "Stream3" are names chosen by the user and are used to identify pairs of connected streams. Note, that these names are of the user's choice and can be replaced by any other strings if required!

The *code*-attribute is used to specify the binary object, which has to be executed for each component. If the *code*-attribute is not used, the CDL-compiler assumes, that the name of the task component matches the name of an executable binary object. To use a binary object, which is not found using the standard search-paths, a full pathname can be specified. If the binary "master", for example, resides in a subdirectory called "/helios/user/private", it can be specified by using the *code*-attribute in the following way:

```
code            /helios/user/private/master
```

Component "Task_A" is forced to be executed on processor "01" inside the subnet "/Cluster". Such a static placement of components can be generated by using the *puid*-attribute. The *puid*-attribute makes particular sense, if the physical network topology is identical with the logical communication topology given with a Task Force and if only <u>one</u> Task Force is to be run in the network. In this case, the step of "mapping" a Task Force automatically onto a given network can be avoided without a loss of "mapping-qualitiy" (10). Another reason for using the *puid*-attribute may be, that a certain component has to be placed on a distinct processor because of the need to access certain features, provided by such a node like bit mapped graphics or other interfaces to the "real world" (11).

The *memory*-attribute gives the *Task Force Manager* useful information to decide, where to place different task components in a network. In combination with the memory-attribute used within resource-maps, the *Task Force Manager* can "look-ahead" and afterwards calculate the number of task components, which can be placed on different nodes, without trying to start them directly by performing the *Execute()* system call. Although the *memory*-attribute is not recommended in general, it should **always** be integrated into component declarations! This is done in case certain processors might be running out of memory, when a Task Force is set up.

(10) ; See chapter 4 for more details about Task Forces and mapping strategies.

(11) : The use of the attribute *attrib* - in combination with similar resource map specifications - does not offer in general the possibility to specify devices, which are associated to a certain processor and are recognized by the system. Therefore the use of this attribute should be avoided!

2) The second example of a Task Force description using the *Component Distribution Language*, also consisting of three components. In this case, we define the communication structure by using pipe-operators within the CDL-script. This makes the specification of streams within different task components obsolete! A CDL-script for such an application - named "test.cdl" - may look like this: (There are bidirectional communication channels created between Task_A and Task_B and also between Task_B and Task_C.)

```
#!/helios/bin/cdl

component Task_A {
      code        master;
      memory      100000;
}
component Task_B {
      code        slave;
      memory      100000;
}
component Task_C {
      memory      100000;
}

# In this case, the communication structure is defined
# by the user, whereas the CDL-compiler is responsible
# for a proper mapping onto streams.

Task_A  <>  Task_B  <>  Task_C
```

(Listing 7)

To get detailed information about the mapping of streams done by the CDL-compiler, a listing of all task components, derived from a CDl-script, can be generated by the CDL-compiler using the -l option. Based on such a listing, the user can make sure, that the right streams are used within his programs. If we assume, that our CDL-script is named "test.cdl", we can generate a listing-file "test.lst" during compilation of the CDL-script by performing:

```
%      cdl  -o test  -l test.lst   test.cdl   <CR>
```

The listing, we have got as a result looks like this:

```
command:

Task_A  <>  Task_B  <>  Task_C

list of components:

name:      Task_C
code:      /Cluster/IO/helios/example/slave
ptype:     ANY
puid:
attribs:
memory:    100000
streams:   <| stream0(0),  >|stream1(1) ,  stderr(2) ,  unused(3)
arguments: Task_C

name:      Task_B
code:      /Cluster/IO/helios/example/slave
ptype:     ANY
puid:
attribs:
memory:    100000
streams:   <| stream2(0), >| stream3(1),  stderr(2) ,  unused(3)
           <| stream1(4), >| stream0(5)
arguments: Task_B

name:      Task_A
code:      /Cluster/IO/helios/example/master
ptype:     ANY
puid:
attribs:
memory:    100000
streams:   stdin(0) , stdout(1),  stderr(2) ,  unused(3) ,
           <| stream3(4), >| stream2(5)
arguments: Task_A
```

If we take a look at the stream connections, we can see that Task_A and Task_B are communicating by using the standard streams (on the side of Task_B) and the first pair of auxiliary streams (on the side of Task_A). The first pair of auxiliary streams is also used by Task_B to communicate with Task_C, which again makes use of the standard streams.

The CDL-compiler verifies the existence of each component specified in a CDL-script. If one binary object can not be found using the standard search-paths, an error message is shown and the compilation is aborted.

A last note: It is not absolutely necessary to make an ultimate decision among these two alternatives; the "manual" and the "automatic" mapping of streams. It is also possible to make use of both approaches within a CDL-script. A typical application example may be a Task Force, consisting of a number of elements which are building an uni-directional pipe (using the "|" operator for simplicity) with some additional streams, allocated explicitly by using the *streams*-attribute.

3.4 Usage of Indices and Parameters

Topics:

- Indices for streams and components
- Run-time parameter
- Compile-time parameter

Writing a CDL-script for a large application tends to become a little bit cumbersome, if each component has to be specified explicitly. The CDL-syntax allows the usage of indexed task components and streams, which makes it much more easier to describe regular communication topologies like uni- or bi-directional pipes or rings. In combination with the possibility to pass compile- or run-time parameters to a CDL-script, it is possible to write compact and versatile CDL-scripts.

The following example CDL-script describes a ring-topology with n elements, which makes use of a pair of auxiliary streams ("6" for reading and "7" for writing) for communication. There is one controller component and a chain of n-1 elements within the communication ring. The last element uses the same binary code as the other elements in the chain with the only difference being, that this component is forced to close the ring by connecting its write-stream with the reader of the controlling component ("control"). It is possible to pass run-time arguments to a CDL-script. They are identified by using the meta-characters $1, $2, ... and so on - a scheme, which is well known from shell-scripts.

```
#! /helios/bin/cdl

component control {
        memory      500000;
        streams     ,,,,,,<|Stream{0},>|Stream{1};
}

component element[i] {
        memory      100000;
        streams     ,,,,,,<|Stream{i+1},>|Stream{i+2};
}

component last {
        memory      100000;
        code        element;
        streams     ,,,,,,<|Stream{$2+1},>|Stream{0};
}

control  $1  ^^ ( ^^ [i<$2] element{i} ) ^^ last
```

(Listing 8)

The names of the inter-connecting streams are built by using the "Stream" text-pattern [12] and the result of the calculations within the curly brackets ({...}). This makes it possible, to define a sequence of chaining elements by using <u>one</u> component declaration. In the example above, the component declaration for "element" is replicated $2-times and the index "i" is used to build unique component-names and stream identifiers. The run-time parameter $1 is directly passed as an argument to the component "control".

If the index itself has to be passsed as an argument to a task component, it has to be pre-fixed by a leading percentage ("%") character. To give the index "i" as a second parameter to "control", the description-line of the parallel application within the CDL-script has to be extended in the following way:

```
control  $1  %i  ^^ ( ^^ [i<$2] element{i} ) ^^ last
```

(12) : ... or any other string of the user's choice!

If it is desired to use a pre-compiled CDL-script, we have to distinguish between compile- and run-time parameters. The terminology used for parameters in pre-compiled CDL-scripts differs slightly in comparison to CDL-scripts, which are executed immediately:

```
1)    "Textual" CDL-script:

Compile-time parameter  =  $n
Run-time parameter      =  $n

2)    Precompiled CDL-script:

Compile-time parameter  =  $n
Run-time parameter      =  \$n
```

Referring to the example above, we have to change $2 into a compile-time parameter, if we want to make use of a pre-compiled CDL-script. The line describing the parallel application within the CDL-script can be modified in the following way:

```
control \$1  ^^  ( ^^  [i<$1] element{i}  ) ^^ last
```

To compile the CDL-script for a fixed topology ring with 12 "element"-components (= a total number of 14 task components), it can be compiled by typing:

```
%     cdl  -o  target  source.cdl  12    <CR>
```

After setting the "cdl"-flag in the actual environment, the Task Force can be executed by calling the pre-compiled binary object. An argument given to this binary is passed as a run-time argument to the "control"-component within the Task Force. These two steps can be performed in the following way:

```
%     set   cdl     <CR>
%     target   16     <CR>
```

As explained in greater detail below (Chapter 4), it is in general to be preferred to start a Task Force by executing the CDL-script directly than working with pre-compiled binary objects, having the cdl-flag set. The main reason for doing this is that - as a side effect of setting the cdl-flag - not only Task Forces, but also single commands (like *cc*, *emacs*, ...), which are executed from a shell, are distributed through the network under the control of the *Task Force Manager*! This can make the system's behaviour under certain conditions unpredictable for the user.

3.5 An Example Task Force

The following chapter gives a fairly simple example of a Task Force, consisting of two components. There is a master component, which generates some data and passes them to a slave component. The slave performs some data-processing operations and sends the results back to the master, where a separate receiver process is forked to get them asynchronously. Both task components make use of the fully-checked POSIX based procedures *full_read()* and *full_write()*, as outlined above. A module with these two procedures has to be linked with the master and the slave component.

The listing of the **master** component:

```
/**********************   master.c   ***********************/

#include <helios.h>
#include <nonansi.h>
#include <posix.h>
#include <stdio.h>
#include <stdlib.h>

void   receiver (void);

/* This semaphore is used for synchronisation of sender */
/* and receiver process at the termination point.      */

Semaphore  sync;

int
main ( int argc , char *argv[] )
{
    int i,n;

    n = atoi ( argv[1] );

    InitSemaphore ( &sync, 1 );
    if ( Fork ( 4000, receiver, 0 ) == NULL )
    {
        fprintf (stderr, "%s: Unable to fork receiver process\n",
                argv[0]);
        exit (1);
    }
    for ( i = 0 ; i < n ; i++ )
    {
        full_write (5, (char *) &i, 4);
        printf ("master: sent data %d to slave\n", i);
        fflush (stdout);
    }

/* -1 is used as a termination identifier. */

    i = -1;
    full_write (5, (char *) &i, 4);
    printf ("master: sent termination word to slave\n");
    fflush (stdout);

    Wait ( &sync );
}

void
receiver ( void )
{
    word rec = 0;

    Wait ( &sync );

    for ( ; rec != -1 ; )
    {
        full_read (6, (char *) &rec, 4);
        printf ("master: got result %d from slave\n", rec);
        fflush (stdout);
    }

    printf ("master: got termination word back from slave\n");
    fflush (stdout);

    Signal ( &sync );
}
```

(Listing 9)

The listing of the **slave** component:

```
/*************************   slave.c   *************************/

#include <helios.h>
#include <stdio.h>
#include <posix.h>

int
main ( )
{
    word rec = 0;

    for ( ;; )
    {
        full_read (4, (char *) &rec, 4);
        printf ("slave: Got data %d from master\n", rec);
        fflush (stdout);

        if ( rec == -1 )
            break;

        /* At this point, we do some processing ... */
        rec = rec * 2;

        full_write (7, (char *) &rec, 4);
        printf ("slave: Passed calculated data %d back to master\n",
                 rec);
        fflush (stdout);
}

    full_write (7, (char *) &rec, 4);
    printf ("slave: Passed termination word back to master\n");
    fflush (stdout);
}
```

(Listing 10)

The **CDL-script** for this Task Force is really compact and looks like this: (The file is called "calc.cdl".)

```
#! /helios/bin/cdl

component master {
      memory      50000;
      streams     ,,,,,>|s0,<|s1;
}

component slave {
      memory      50000;
      streams     ,,,,<|s0,,,>|s1;
}

master  $1   ^^  slave
```

(Listing 11)

To start the Task Force by executing the CDL-script, the following line may be executed. (For this example, a sequence of 50 numbers is generated by the master and passsed to the slave for processing.)

```
%      calc.cdl  50  <CR>
```

3.6 Using a Load Balancer

Topics:

- Farming constructs and Load Balancing
- Limitations of usage

The use of the *farming*-construct (|||) within a CDL-script seems to be one of the most attractive features of the *Component Distribution Language*. In combination with the general replicator ([]) and a *load balancer* task, which build a transparent interface between a master task and a number of replicated slave-tasks, it is very easy to describe Task Forces which distribute data among various slaves for processing. Unfortunately, the use of the *farming*-construct in combination with the *load balancer* places some restrictions on the user, which have to be taken into account to make best usage of the available resources:

- Firstly, the user has to be aware, that the *load balancer* allocates various chunks of memory dynamically for buffering of packets, which are transferred between the *load balancer*, the master task and the slave tasks. This makes it essential, to place the *load balancer* on a processor which is equipped with enough memory to do this job. Other components of a Task Force should not be executed on this node!

- The *load balancer* as supplied under Helios 1.15 supports only *packet*-mode. Therefore, the application has to make use of proper defined packets for data transfer. An example Task Force, which is based on this *packet protocol* can be found in Appendix B.

- The following fixed pairs of streams are used by the *load balancer*: The master communicates with the *load balancer* by using the first pair of auxiliary streams (4 and 5). The slaves make use of the *stdin* and *stdout* streams to exchange data with the *load balancer*.

The user can write his own *load balancer* task, if he pays attention to the interfaces, which have to be defined for the communication with the master and an arbitrary number of slaves. To give an idea of how a *load balancer* is structured, the Helios *load balancer* is supplied with its source code on the distribution medium. The source codes gives also some hints about special features, like "*broadcasting*" and *terminate* function codes.

It should be explicitely mentioned at this point, that working with a *load balancer* does not offer under all circumstances the best results in performance and usage of resources. Such a centralized instance may become a kind of bottleneck, if the number of slaves is drastically increased! Therefore it has to be considered carefully, under which conditions the use of a load balancer makes sense.

3.7 Language Dependent Notes

Topics:

- Task Forces written in FORTRAN-77
- Task Forces written in Modula-2 and in Pascal

The Helios FORTRAN-77 compiler has implemented an interface to POSIX-compatible stream operations. The FORTRAN programmer can directly make use of these calls by including them in the FORTRAN source code. The POSIX calls are pre-fixed by a "POS_" and offer the same functionality as their C equivalents. A consequence of this is, that the user has to check return-codes for errors to make sure that all data were transmitted correctly (13). A typical POSIX-write operation within a FORTRAN-program may look like this:

```
C    *** USAGE OF POSIX WRITE OPERATION ***
C
     NUM = 128
C
     ERR = POS_WRITE (6, NUM, 4)
     IF  ERR .NE. 4  THEN
C
C    *** ERROR HANDLING CODE ***
C
     END IF
C
C
```

(Listing 12)

The stream number ("6"), the name of a variable or an array ("NUM") and its size measured in bytes ("4") are used as parameters.

(13) : An example for a FORTRAN POSIX-interface with return-code checking can be found in Appendix A. This separate module "posixop.c" defines the procedures *PSX_READ()* and *PSX_WRITE()*, which are written in C and have to be linked with a FORTRAN module.

A general note concerning the usage of Helios FORTRAN within Task Forces: Because it is not possible to generate separate processes within a FORTRAN program and to have the benefit of full asynchronous processes for data transfer, the user has to take great care in defining synchronous data transfer mechanisms to avoid wasting of time and incalculable delays of data flow.

Prospero-Pascal and Rowley Modula-2 are both equipped with an interface to the POSIX-library which can be used for Task Force communication. Please take a closer look to the associated documentation for a more detailed description. Prospero Pascal also supports processes within a Helios task, which makes it possible to work with asynchronous communication models similar to C. The different points concerning Task Forces we have mentioned before, are also relevant in case of applications written in Pascal and Modula-2!

4. Task Force Management

Task Force Management services, based on sequential programming languages and the *Component Distribution Language* for the definition of parallel applications are both unique features offered by Helios and make this operating system especially sympathetic for the parallelization of a wide range of applications.

A separate Helios-server, called *Task Force Manager* (TFM), is responsible for setting up a parallel application ("Task Force"), by creating the stream connections for communication and for monitoring the execution and termination of the different task components within a Task Force. The *Task Force Manager* has a close relationship with the *Network Server* and is not able to run without this second "global" server working in the background. Both the *Network Server* and the *Task Force Manager* have to be established on the first node of the network, called the *root processor*! It should be mentioned, that the *Task Force Manager* under Helios 1.15 has been totally rewritten from scratch and no longer places restrictions on quantities such as the number of processors, task components or streams, which can be managed!

The following chapter is intended to give a better insight into the basic mechanisms used by the *Task Force Manager*; something which is still a little "mystic" to many Helios users. There are a lot of things which have to be taken into consideration while working with Task Forces to make best usage of the system and to avoid problems during the execution of Task Forces.

4.1 The Basic Mechanisms

Topics:

- Getting diagnostics with *diag_tfm*
- The different phases of Task Force Management
- A closer look to the mapping, done by the *Task Force Manager*

Experience with the *Task Force Manager* under Helios 1.1 has shown, that the user-interface - especially the error handling mechanisms and the error messages given by the *Task Force Manager* - are not of great practical use. Nearly all error conditions are reduced to the well-known messages "*Posix Error 22*" and "*Exec Format Error*", which definitely prove, that a Task Force could not be set up successfully, but do not give any hints about the reason for the failure. To get better diagnostic informations concerning the different phases of Task Force Management, a utility *diag_tfm* is added under Helios 1.15. This little program sends messages to the *Task Force Manager* and allows enabling and disabling of different protocol phases, beginning with the step of Task Force creation up to the termination phase (14). To get the initial menu from *diag_tfm* with a list of the selectable options, just type:

```
%      diag_tfm    <CR>
```

As a "short-key", it is also possible to call *diag_tfm* in a non-interactive mode with the arguments *none*, *most* or *all*. Using *none* disables all diagnostics messages, whereas *all* does the opposite. To enable a standard set of diagnostic messages given by the Task *Force Manager* (except: Phase 2 mapping and environment info), just execute:

(14) : The diagnostic output is written on a PC's server window or on the actual active window shell on a SUN, from which *diag_tfm* was executed.

```
%     diag_tfm   most   <CR>
```

(Appendix C presents the output of the *Task Force Manager's* diagnostic report for the example Task Force described in chapter 3.5, based on the option *most.*)

If a Task Force is initiated, it is the first job of the *Task Force Manager*, to "map" the given application onto the network topology to find a nearly optimal placement of all components of the Task Force. (A more detailed description of the mapping process is given below.) After this mapping is done, the Task Force Manager takes the generated mapping data structures and begins to execute the task components remotely on the different target processors.

For each component, a separate monitoring process is created by the *Task Force Manager* to keep control over the behaviour of the whole Task Force by monitoring the work of each component. With *GetProgramInfo*-calls, periodically sent from each monitor process to the associated task component, the *Task Force Manager* gets all information needed. This is in particular useful, if one of the task components terminates because of the occurrence of a certain error condition. If the *Task Force Manager* recognizes this, it is possible to force the correct termination of the whole Task Force.

During setup of a Task Force, the connection of streams is done by using pipe-servers, which are installed on demand on different nodes. The communication streams are created as separate entries in the different pipe-servers directories. After passing an environment to each task component and creation of a new entry in the *Task Force Manager's* directory, the Task Force is established and can begin to run.

To get an survey of the different Task Forces actually controlled by the *Task Force Manager*, the *Task Force Manager's* directory can be listed by using the *ls*-utility. Please note, that precompiled CDL-scripts are entered into this directory with their full names, whereas CDL-scripts - directly executed by the user - build "synthetic" names, based on the string "cdl.tf." with an unique number for identification added.

```
%      ls     /tfm       <CR>
```

The *Task Force Manager* performs **two separate phases** of "*mapping*" to find a nearly optimal placement of task components on the different nodes of a given Transputer network. These two phases of the mapping process are built up in the following way:

1) **Phase 1** of the mapping process is fairly simple and is based on an analysis of the resources, which are available within a Helios network. During this phase, the *Task Force Manager* verifies especially the availability of memory to execute a Task Force, built of a certain number of task components. To assist the *Task Force Manager* doing this job, it is recommended to specify the memory on each node in the resource map and to declare the amount of memory needed by each component of a Task Force by using component declarations within a CDL-script. If this is done by the user in a proper way, the *Task Force Manager* can "look ahead" to decide, how to place a number of task components on the different processors to make best usage of the resources. Other aspects like the network topology, the communication structures within the Task Force and the actual workload of distinct processors are not taken into account at this moment!

An easy example for this first step of the mapping process: Imagine a network as described in figure 3. There are four processors declared as HELIOS-nodes, which can be used for the execution of Task Forces, each of them equipped with 1 MBytes. If we want to run a Task Force, consisting of eight similar components - each of them requiering 200 KBytes of RAM - the *Task Force Manager* will decide to place two components on each node. This is done without any regard to topology dependent aspects and pre-defined communication structures in the corresponding CDL-script! The *Task Force Manager* works straightforwardly by using the *Network Server's* directory and accesses the processors in the same sequence as they are noted there.

At first, it seems to be unsatisfactory to work with such a simple approach. On the other hand, the results of the mapping process are not as bad as might be supposed, because most of the time, there are really simple and regular processor- and communication topologies used, like pipes, rings and grids. Under these circumstances, the mapping results, after finishing phase 1 are usually acceptable.

2) To improve the mapping of irregular Task Force structures, a second phase of the mapping process can be optionally enabled for the *Task Force Manager*. This "**phase 2** mapping" takes also topology dependent aspects into account and tries to find an optimal placement of task components on the basis of the calculation of correlation matrizes. This is a highly heuristic procedure which has advantages, but also some disadvantages, which we mention below:

- The amount of time needed to map a certain Task Force description onto a given network increases with the number of processors and the number of task components in an exponential way. This can result in extraordinary long times for setting up larger Task Forces (15).

(15) : For example: The "phase 2 mapping" of a 20 processor pipe onto a 5x4 processor grid needs nearly one minute to finish!

- In theory, the overall quality of the phase 2 mapping is nearly optimal, if all possible combinations of network and communication topologies are taken into account regardless of whether they make sense or are of practical use. If we concentrate on examples, which have a practical relevance, the results of the mapping process are only sub-optimal; especially if we take the time for performance of the needed calculations into account.

As a consequence of this, the phase 2 mapping is by default **disabled** at the moment, when the *Task Force Manager* is started, The user can enable phase 2 of the mapping process by using the *diag_tfm* - utility as described above. To become totally free from the mapping algorithms used by the *Task Force Manager*, it is possible to specify a processor attribute (*puid*) for each task component within a component declaration. As mentioned before, this makes particular sense, if the topology of the network and the communication topology within the application are identical.

4.2 Working with Task Forces

Topics:

- Working with or without the CDL-flag ?
- Task Force termination
- The problem of memory fragmentation

Working with a Helios shell offers the possibility, to set a **flag** called "**cdl**" for Task Force execution. In contrast to the basic variant, to make use of Task Force Management services by executing CDL-scripts directly, every single binary object which is executed is treated in this case as a separate Task Force and is distributed in the network.

- This can make especially sense in a "**development** environment", based on a small network size (begining with two up to four processors), because different shells, compilers and other utilities are transparently placed on different processors. Under these circumstances, setting the cdl-flag results in a better utilization of the processor resources.

- The use of the cdl-flag becomes critical, if a network is intended be used for development purposes and for **application** execution in parallel. In this case, it is desired to keep parts of the network (usually the larger part) free for the exclusive execution of these "application Task Forces", which shall not be influenced by other jobs, meanwhile initiated by the user. This cannot be achieved, if the cdl-flag is set, because there is no internal barrier between processors within a subnet. It may happen for example, that the *Task Force Manager* places different utilities temporarily on processors, which are required with their full memory space for the execution of application Task Forces. Because the behaviour of the system is in this case totally unforseen, it is better to make no use of the cdl-flag!

If the cdl-flag was previously set, it can be removed by typing:

```
%       unset    cdl      <CR>
```

If more than one processor is required for development purposes, it is a good approach to declare not only the root processor, but also some other nodes in the resource map as SYSTEM-nodes and to start different shells and other utilities explicitely on these Transputers. The remaining processors - declared as HELIOS-nodes - are for exclusive execution of application Task Forces and are not directly influenced by jobs executed on the set of "development processors" (16).

In general, Task Forces can be **terminated** in two different ways: Simply by typing CTRL-C from the corresponding shell or by removing the associated entry from the *Task Force Manager's* directory. If we have started the job in the background, we have to bring it first back into the foreground to terminate it via CTRL-C. Imagine a Task Force running in the background with the job identification number "4". By typing

```
%       %4    <CR>
```

for example, we can move this job back into the foreground again and terminate it subsequently by typing CTRL-C.

(16) : A last note: If the cdl-flag is **not** set, Task Forces can only be started by executing their script-files directly from a shell. If a precompiled CDL-script is to be executed, the cdl-flag has to be set!

To pass a termination request directly to the *Task Force Manager*, a message can be sent to this server, referring to an entry noted in the *Task Force Manager's* directory by using the standard *rm*-utility. An example: If we want to terminate a Task Force named "cdl.tf.1.4", we can execute the following command line:

```
%      rm      /tfm/cdl.tf.1.4        <CR>
```

The *Task Force Manager* tries very hard to get rid of task components within a Task Force: If a termination condition arises, it sends a KILL-signal to each of the task components. If this does not succeed, it deletes them explicitly.

An important note for FORTRAN-programmers: The underlying FORTRAN run-time system is actually not able to manage direct termination requests, which are passed to the *Task Force Manager* directly by using *rm*! Therefore it is strictly recommended to terminate FORTRAN Task Forces only by using CTRL-C within the associated user-shell!

Another point, which has to be taken into consideration is the general problem of **memory fragmentation** on processors like Transputers, which are not able to manage virtual memory. There is a great possibility that previously contignous memory will become increasingly fragmented during a Helios session as tasks and Task Forces are executed and terminated.

A realistic scenario can make this problem a little bit more clear: There are certain Helios servers like *pipe*, *ram*, *null* and *fifo* and resident libraries like *CLib* and *Posix*, which are not installed by default on every node to keep as much memory as possible free for application programs. These parts of code are loaded "on demand". Now imagine a Task Force, which is started directly after the system has booted. At this time, none of the servers and libraries mentioned above are installed. During Task Force setup, the different task components are placed on the distinct processors with regard to the results of the mapping process. After this is done, pipe servers are required to establish the necessary streams for communication. Because pipe servers are requested at this moment for the first time, they are loaded into memory and are placed on the next free memory location. After finishing the Task Force, all memory used by the different components is freed. The pipe servers on the other side still remain in memory, staying at their initial location. This results in the first significant fragmentation of memory! If another Task Force is initiated while the first one is still running and some of the components require the X-Library for example, this resident library is also loaded on demand and placed at the next free memory location and so on ...

The problem of memory fragmentation on the Transputer is not Helios specific and cannot be solved in general in an efficient way because of the lack of assistance from the Transputer hardware. But there is a possibility to moderate the increase of memory fragmentation by "pre-loading" certain standard servers and resident libraries at an early stage directly after the system has booted. A utility called *preload* is added under Helios 1.15 to do this job. This program can be executed with the name of a subnet or a list of processor names specified as arguments. The following servers and libraries are installed:

Option:	(none)	-all
CLib	XX	XX
Posix	XX	XX
ram		XX
null		XX
fifo		XX
pipe	XX	XX

(Figure 5)

To perform a preload of the minimal set of servers and libraries for all processors of a subnet named /Cluster, the following command-line may be executed:

```
%      preload   /Cluster   <CR>
```

As an alternative to this, it is possible to pass a list of processor names to build a subset for pre-loading. To pre-load the complete set of servers and libraries as listed above on five distinct node, the command

```
%      preload   /01  /02  /03  /04  /06  -all   <CR>
```

may be used.

The *preload* command can be integrated into the loginrc-script file, if it is desired to work with pre-loaded libraries and servers every session.

5. Network Mechanisms

Helios 1.15 uses basically the same *Network Server* as Helios 1.1 with some modifications and enhancements concerning the boot-mechanism and the usage of links. The new Helios bootstrap is nearly five times faster than the old one and does not impose any restriction on the desired topology. The two main difficulties concerning the Helios 1.1 boot mechanisms - working with meshed topologies and rebooting active Helios networks without a centralized reset - does not cause problems anymore.

5.1 Different network sizes

Topics:

- Choosing the right memory size for the root node.

There is a special point, which has to be taken into early consideration at the moment, when a Helios multi-Transputer network has to be established: The size of the root node's memory. The *Network Server* and the *Task Force Manager* make wide usage of dynamic memory, which is allocated depending on the size of the network and the number of Task Forces and Task Components actually running. Because of the fact that the *Network Server* and the *Task Force Manager* have to be placed nearly always on the root processor, it is desirable to have as much memory as possible available on this node to avoid any kind of memory bottlenecks during operation.

The following basic scheme gives a guideline on choosing the right amount of memory for the root processor. There are two different categories defined, which differ in the way how the root node is used during work with the system:

1) **Category A** describes an environment, which keeps, for most of the time, enough memory on the root processor to run for example an additional shell or to execute some other utilities.

2) Under **Category B**, the root node's memory may be fully used by the *Network Server* and the *Task Force Manager.* This can make it impossible, to run any other program on it!

In general, the root node should always be declared as a SYSTEM node to keep it free from Task Force components. This is absolutely recommended, if you are using a configuration of category B, because otherwise the system may come into a critical state while running out of memory resources.

The table of root processor memory sizes, depending on the number of processors being integrated in a network looks like this:

Root node to be equipped with ...

Processors	Category A	Category B
16	1 MB	< 1 MB
32	2 MB	1 MB
64	4 MB	2 MB
128	> 4 MB	4 MB

(Figure 6)

5.2 Booting a Multiprocessor Network

Topics:

- Starting the *Network Server* within the initrc-file
- Starting the *Network Server* from a user shell
- The Helios 1.15 boot-path

Every Helios network contains exactly one processor which is directly connected to the host computer's bus via a "bus-bridgehead". This root-processsor is booted under the control of the server program running on the host. An arbitrary number of network processors is booted by the *Network Server* routines from the moment the *Network Server* is successfully established on the root-processor.

Starting a *Network Server* is usually done by a special system task called *init*, which becomes active directly after setting up the first node and before allowing any user interaction. This task reads an *initrc* script-file, which is searched in the etc-directory and executes the commands described there one after another. The following two lines within the standard *initrc*-file cause the start of the *Network Server* and the *Task Force Manager*. The *init* task proceeds after the successful setup of the *Task Force Manager* (17).

```
run -e /helios/bin/startns startns -r /helios/etc/default.map
waitfor /tfm
```

The order of commands in the *initrc*-file has influence on the sequence of the system's setup steps: Under some conditions, a certain sequence may be strictly recommended. For example: If you want to run your X-server on a graphics node like Parsytec's GDS board, which might not be directly conected to your

(17) : Depending on the user licence, the program "startns.any" may be executed instead of "startns". In this case, no limitations on the number of processors are set.

root-processor, you are forced to start the *Network Server* first to boot the network and then run the terminal emulator program, which accesses the X-server. If you do it the other way round - as shown in the 1.1 release initrc example file - the X-server may be searched before the graphics node is succesfully initialized. This causes problems under certain circumstances (18).

It is not essential, to start the *Network Server* from the *init*-task. It may be desirable under some conditions, to boot only the root-processor during a first step - without using any resource map - and then starting the *Network Server* from a user shell as a totally independent second step. This makes particular sense if the user has access to a machine like a Parsytec MultiCluster or SuperCluster, which allows dynamic partitioning and configuration from a running Helios network. In this case, the user can first allocate a set of resources and configure them by using the electronic configuration facilities. If this succeeds, it is possible to start the *Network Server* explicitly with a resource map binary as an argument, describing the actual configuration and boot the processors subsequently. If the allocation has failed because of a lack of resources, the user can try again with a network of a smaller size.

If a *Network Server* has been established, the system has to be rebooted, if another resource map, describing a different topology is to be used. This is recommended, because the *Network Server* is not able to perform a context switch from one Helios network topology to another. Rebooting the whole network and starting the *Network Server* again forces this context switch!

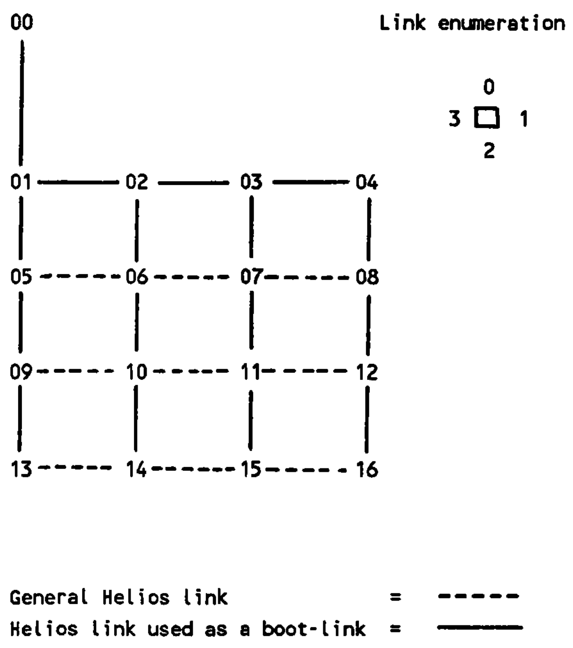
Imagine a topology consisting of a root processor and a network with 16 processors, configured as a 4x4-grid. The resource map is kept in a file called "grid4x4.map". After booting the root node and (eventually) allocating the resources, the network can be booted from a user shell by using the following command-line:

(18) : Please note, that this is only a representative example - there may be other system configurations, which also require a rearrangement of statements within the *initrc*-file.

```
%     startns  -r  grid4x4.map   <CR>
```

The bootpath of Helios has changed from a "depth-first" proceeding in version 1.1. to a "breadth-first" mechanism under Helios 1.15. The breadth-first mechanism has some advantages in comparison to the old one, especially resulting in a much more faster bootstrap with a better balanced boot tree.

The bootpath, used to establish a 4x4 processor network, looks like this:



(Figure 7)

As mentioned during the discussion of resource map structures before, it is strictly recommended to choose an order of processor declarations which guarantees, that a new declared processor was successfully booted by another processor, before it is accessed the first time!

The Helios 1.15 boot mechanism takes care, that all link connections, which are declared in the resource map are usable for message passing. This is done by turning the associated links into *intelligent* mode with the state *running* after successful booting of a processor (details see below). It is in general of great advance, to use a meshed topology, because in this case, it is possible to communicate by using various redundant link paths which keeps the individual link traffic rates as low as possible. Another aspect, that gives a good argument for the usage of redundant link paths, is the greater fault tolerance.

5.3 Reset and Reboot

Topics:

- Reset and reboot using the Network Context
- Reset and reboot with Parsytec hardware

There are two different approaches to reset and reboot a particular processor or a part of a Helios network:

The first variant takes the informations hold by the *Network Server* and allows a selective "soft" reset and reboot of a processor under the control of the *Network Server*. This mechanism is totally independent of the underlying hardware topology and the initial reset and boot mechanisms, but requires some additional informations about the processors from the *Network Server*. With the two utility programs *reset* and *boot*, a selective reset and boot can be performed.

To reset a node called "04" and boot it again afterwards, the following command sequence has to be executed:

```
%    reset  /04    <CR>
%    boot   /04    <CR>
```

An important restriction to be mentioned for this approach is, that it is neither possible to extend a network by booting "new" processors via links which have to be specified explicitly, nor is it possible to reset a node physically via a dedicated link!

Parsytec Transputer hardware differs from all other vendors' products in the way, that it is allowed to reset every processor via each of the four links. No centralized reset mechanism is used. This results in a functionality which makes Helios especially sympathetic to Parsytec hardware: Without terminating a Helios session, it is possible to reset and reboot nearly every processor in a network through a dedicated link without affecting the others. Beyond this, processors can be integrated into the network context during a Helios session without declaring them in a resource map; just by resetting and booting them via links which are known by the user! This results in a system of maximal flexibility with a very high fault tolerance, because the crash of one processor usually does not have major influence on the behaviour of the remaining nodes (19).

To get the full benefit of using Parsytec hardware, a special binary object for individual reset and boot of a processor is supplied. This file, called *pa_rboot*, can be found in the lib-directory and should replace the original *rboot* program, if Parsytec hardware is used. The *Network Server* uses the *pa_rboot*-binary during the initial network boot to reset and boot all processors which are found, following the boot path as shown above.

The *pa_rboot* program can also be called directly from a user shell to reset and reboot a certain processor which may have crashed. This can be done as shown below: We assume, that we want to reboot a processor called "05" from another node which has given the name "04". The link to be used may be link number 2. The actual subnet is named /Cluster. The following command can be executed for resetting and rebooting node "05".

```
%     /helios/lib/pa_rboot  2  /04  /Cluster/05   <CR>
```

(19) : The only processor which should never be reset and rebooted during a Helios session is the root-processor.

If you only want to reset a certain processor, you can use the *pa_reset* utility and execute it remotely on a neighbour node. With reference to the example above, this can be done by typing:

```
%     remote  04  pa_reset  2    <CR>
```

As mentioned before, resetting and rebooting processors by using *pa_reset* and *pa_rboot* is not restricted to nodes which are declared in a resource map. It is also possible, to boot additional nodes during a Helios session and expand a network step by step. These nodes can be used nearly in the same way as processors which are booted under the control of the *Network Server* with the only major difference, that the *Network Server* is not able to integrate them into its actual context. This results in a behaviour under which the *ls* command on this processor (for example: "ls /NEW/tasks") works very well, whereas *ps*, which accesses the *Network Server's* directory fails. It is also impossible for the *Task Force Manager* to make use of such a node during initialization of a Task Force, because the *Task Force Manager* also depends on the information, kept by the *Network Server*. All other operations - especially distributed searches for programs running on such a node - will succeed.

5.4 Usage of Links

Topics:

- Link modes: *dumb* and *intelligent*
- Usage of *dumb* links
- Combining Helios with OCCAM networks
- Usage of redundant link paths

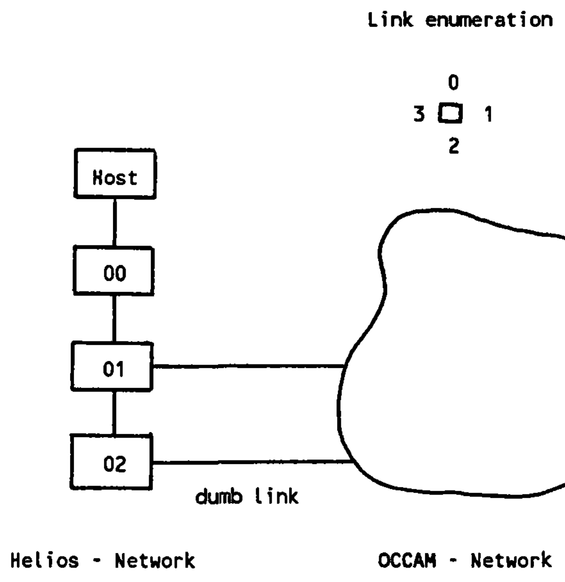
The Helios kernel routines consist of processes called *Link Guardians*, which are responsible for managing all link data transfer operations for each individual physical link connection. Helios distinguishes between two different "modes" for Transputer links (20), *intelligent* and *dumb*. A link used for standard Helios message passing communication is configured into *intelligent* mode. A set of various states like *running*, *dead*, *crashed* and *timeout* is associated to an *intelligent* link. If a link is to be kept free from all Helios message passing protocols, it has to be turned into *dumb* mode. In this case it is possible to perform basic byte-to-byte link communication between two nodes without any effects from higher Helios communication mechanisms.

The user can take control over the link modes by using the *Configure()* system-call. In conjunction with the *LinkData()* call, which gets a copy of the actual configuration vector for the desired link, it is possible, to change the mode of a link during a Helios session. This can be especially useful, if a Helios network is to be combined with an OCCAM network (= "hybrid network"). In this case, the communication between the two networks has to be managed from the Helios side by using a *dumb* link. The system calls *LinkIn()* and *LinkOut()* are supplied to perform data transfer operations on a *dumb* link. The communication protocol between OCCAM and Helios is full under the control and responsibility of the user. The second scenario, which makes use of different link modes and associated states, is a distributed Helios network of an arbitrary

(20) : We do not take account on non-connected links here.

number of processors, organized in pools of various sizes and topologies, which are connected and disconnected at any time during a Helios session and allow data transfer and remote execution of Task Forces and single tasks between different logical networks.

The following configuration gives an example of a "hybrid network", consisting of OCCAM and Helios nodes. We assume, that there is a MultiTool server (21) running on the Helios node "01", using link number 1 as an entry into the OCCAM subnetwork. This link and link 1 of processor "02" are configured into *dumb* mode to allow direct link data transfer between the two networks. (These link have to be declared as "external" within the Helios resource map.)



(Figure 8)

(21) : MultiTool is the Parsytec specific Transputer development environment, which is based on the Inmos TDS.

Such an environment makes special sense, if a time-critical application has to be realized. Because of the fact that Helios - as a non-deterministic operating system - is not able to guarantee fixed response times which are essential for real-time applications, it is a good approach to use a "hybrid network": On the "OCCAM-side", all time-critical parts of the application are handled including the access to high-speed devices. The Helios part of the network can be used for comfortable user interaction (X-Windows), fast filing system (Helios filing system with SCSI-devices) and all other tasks concerning data-processing and number crunching. On the Helios side, the user gets the full benefit of a comfortable development environment and standard programming interfaces.

The following fragment of code demonstrates the usage of the *LinkData()* and *Configure()* system calls to turn a link into *dumb* mode. The configuration vector of the specified link is read and whether the link is in a running state is checked. If this is the case, nothing is changed, because another instance makes use of the link. Otherwise, the link is put into *dumb* mode.

```
#include <helios.h>
#include <link.h>
/* ... */

{
        struct  LinkInfo  linfo;
        struct  LinkConf  conf;
        word    link_num;
    /* ... */

        LinkData ( link_num, &linfo );
        if ( linfo.State != Link_State_Running )
        {
                conf.Id    = link_num;
                conf.Mode  = Link_Mode_Dumb;
                conf.State = 0;
                conf.Flags = 0;

                Configure ( conf );
        }

/* ... */
}
```

(Listing 13)

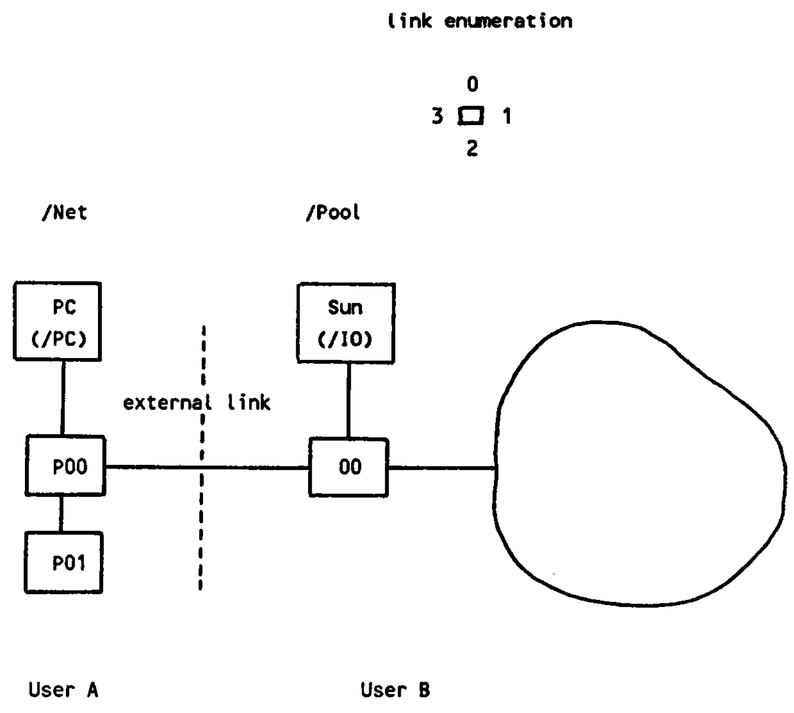
5.5 Basic Multi-User Facilities

Topics:

- Connecting and disconnecting Helios networks
- Naming conventions
- "Application shells" and remote Task Forces
- The question of security

Helios 1.15 offers basic mechanisms and tools to connect and disconnect various Helios networks at runtime. External links, declared in different resource maps build the interface between these different subnetworks. Because it is sensible to establish and destroy logical connections between separate Helios networks and to share processor resources in common, the basic mechanisms and some well-proved results of multi-networking will be discussed in greater depth below.

At first, we define a sample test environment with a practical working surface to point out all relevant aspects of multi-networking: There are two separate Helios networks, one consisting of three processors (User A, subnet name "Net"), the other one consisting of 16 processors (User B, subnet name "Pool"). User A uses a PC as a host whereas user B resides on a Sun. The link connection between the two networks via the terminal nodes /Net/P00 (link 1) and /Pool/00 (link 3) is declared in both resource maps as an external link.



(Figure 9)

Helios handles these external links in the following way: After booting up the two separate subnetworks "Net" and "Pool", the external links are configured to *intelligent* mode and reside in *dead* state [(22)]. They can be activated from both sides by using the *elink* utility program. If user A, for example wants to come into contact with user B, he has to execute the *elink*-command like this:

```
%    elink  /00  1   <CR>
```

After this, the links on both sides have changed to *running* state and allow distributed searches through the whole network. To access the Network Server of the subnet "Pool" from the side of user A for example, the command

(22) : This can be verified in a very easy way by using the extended map utility under Helios 1.15 with the submenu for link activities.

```
%      ls -l /Pool/ns    <CR>
```

may be used. Another example: If user B wants to copy a sub-directory called "/helios/example" from the host's filing system of user A (/PC) into a sub-directory "my_exam", he can execute the following command line:

```
%      cp /PC/helios/example/*  my_exam
```

The connection between two subnetworks can be removed by using the *dlink* utility. This turns the associated link into *dumb*-mode, which makes it impossible for tasks "from the other side" to send any Helios message over this link. Unfortunately, the reconfiguration of a link into *dumb*-mode causes the *Link Guardian* on the other side also to change the link into *dumb*-mode. This makes it in general impossible to re-establish the connection from one side again, if a link has once entered *dumb* mode. As a solution, a utility program *intellgt* is supplied under Helios version 1.15, which can be used to keep a link *intelligent*. This program has to be called by the user B in the following way [23]:

```
%      remote  00  intellgt  3   5  &     <CR>
```

(23) : In this case it is assumed that user A accesses the processor pool of user B for operating. This makes it essential for user B to keep the external link (3) intelligent to allow user A connection and disconnection.

The task *intellgt* is installed on processor "00", which "watches" on link 1 (first parameter) of this Transputer and - if the link has turned to *dumb* mode - reconfigures the link back to *intelligent*. This is done in steps of five seconds (second parameter). If such a task is installed on processor "00" (in "Pool"), user A is able to connect and disconnect the two subnets without any restrictions! In the worst case, user A has to wait for maximal five seconds to satisfy his request for connecting both subnetworks.

There are some other critical points which have to be taken into consideration to avoid name clashes:

1) The I/O-nodes should be given individual names to allow an unique identification of the different host-machines and the devices controlled by them. This is done in the example configuration by renaming the "Net"-I/O node to "PC" instead of "IO".

2) By default, the system login-windows are given the name "console". This may also cause problems during distributed searches, finding the "right" console-device for terminal output. Therfore it is a good practice, to give the initial console window other names than "console". This can be done by changing the appropriate line in the *initrc*-file:

```
console  /window  console_2
```

This line causes the creation of login window-shell with the name "console_2" instead of "console" which is used as a default value..

3) Sometimes it may be useful to give each processor within the whole network an unique name. This guarantees, that there are no problems arising concerning context ambiguities, while searching for a particular processor.

There are in general two possibilities to make use of processor resources from remote subnetworks: At first, a utility program called *remotetf*, which is added under Helios 1.15, can be used: This little program can be compared with the standard *remote*-utility with the major difference, that it accesses a *Task Force Manager* across a network boundary instead of a processor within the local network context. If user A wants to make use of the processor resources of user B, he can use *remotetf* in the following way: (A task force named "test.cdl" is passed to the *Task Force Manager* /Pool/tfm to be executed.)

```
%      remotetf  /Pool  cdl  test.cdl    <CR>
```

It should be noticed, that it is only possible to pass executable binary objects as an argument to *remotetf*. Therefore, the CDL-compiler is passsed as the main argument for *remotetf* and the CDL-script and optional parameters follow as parameters of the CDL-compiler. If the connected subnetworks are using for example identical I/O-node or processor names, it is a good practice, to specify full pathnames for *puid* and *code* attributes within the CDL-script to avoid name clashing conflicts. A CDL- script, named "test.cdl", may look for example like this:

```
#!/helios/bin/cdl

component master {
     memory  100000;
     code            /Net/PC/helios/test/master;
     }

component slave {
     memory  100000;
     code            /Net/PC/helios/test/slave;
     }

master [$1] ||| slave
```

(Listing 14)

The second variant for user A to work with the processor pool of user B is, to place a shell - in this case called "application shell" - remotely on one of the processors of the network of user B. Such an application shell can be generated in the following way:

```
%      wsh     /Pool/10        <CR>
```

If user A switches to this new created window shell, he gets the full context of the network "Pool", whereas the terminal I/O streams are redirected to his console device. Such a shell is called "*application shell*", because it is especially helpful to make use of such a shell to start Task Forces and other single task applications. In contrast to this, other shells of user A which reside locally in network /Net can be used for development purposes like editing and compiling ("*development shell*").

A point of general interest is the question of security in such a Helios multi-user environment: Helios 1.15 does not support any mechanisms for access right setting and checking on network level (24)! It is under the full responsibility of the different users to take care, that no conflicts are arising while using a shared processor pool. This makes it especially preferable to work with the *remotetf*-utility instead of using remote installed "application shells", if a better separation of different user activities is to be realized. Nevertheless, it is in general possible for any user to get influence on the behaviour of other user's Task Forces, if they are sharing the same resources!

(24) : Full access right mechanisms will be implemented in the next release of Helios.

Appendix A - Further Information

There are some topics which have to be mentioned here, because they were not discussed in depth before:

- This application guide is only the first part of an additional set of Helios documentation supplied by Parsytec. A second volume, dealing with a wide range of themes which are left out so far, like

  > building resident and non-resident libraries
  > working with dumb-links,
  > communicating via message passing,
  > tayloring the system in general and
  > making best use of utilities and compilers,
  which are Helios specific,

will follow.

- There are some more additional utilities on the distribution medium like *where* and *pp*, which are not discussed in greater detail here. An *emacs* configuration-file called *emacs.rc* (for the PC) is also supplied.

- System programmers can get more information from the "Helios Technical Reports", published by Perihelion Software Ltd. A list of the available items can be received.

Proper POSIX handling under FORTRAN

```
/***************************************************************************
 *
 *          EXTENDED POSIX STREAM-I/O SUPPORT FOR HELIOS FORTRAN
 *       ======================================================
 *
 * o Extensions to the functionality of some POSIX compatible stream-
 *   calls to offer a more comfortable data transfer handling for the
 *   FORTRAN programmer.
 * o This module has to be compiled with the stack checking option
 *   disabled ("c -Fs ..."), if it has to be linked with a FORTRAN
 *   program.
 *
 ***************************************************************************
 * Written  : 8/12/89 , H-J Ermen
 ***************************************************************************/

#include <helios.h>
#include <posix.h>


word    *PSX_READ (word *para[]);
word    *PSX_WRITE (word *para[]);


/***************************************************************************
 * CONTROLLED READ FROM A POSIX STREAM
 *
 * Parameters : para = Pointer to an array of pointers with the
 *                     following meaning:
 *              para[0] : Stream
 *              para[1] : Pointer to the buffer to be transferred
 *              para[2] : Amount of data (bytes) to be transferred
 * Return     : 0 if no error occurrs, otherwise -1
 *
 ***************************************************************************/
word *
PSX_READ ( word *para[] )
{
        word stream, amount, result, ret;
        char *buff;

        stream = *para[0];
        buff   = (byte *) para[1];
        amount = *para[2];
        while ( amount > 0 )
        {
                result = read (stream, buff, amount);
                if ( result <= 0 )
                {
                        ret = -1;
                        return &ret;
                }
                amount -= result;
                buff = &(buff[result]);
        }
        ret = 0;
        return &ret;
}

/*************************************************************************
 * CONTROLLED WRITE TO A POSIX STREAM
 *
 * Parameters : para = Pointer to an array of pointers with the
 *                     following meaning:
 *              para[0] : Stream
 *              para[1] : Pointer to the buffer to be transferred
 *              para[2] : Amount of data (bytes) to be transferred
 * Return     : 0 if no error occurrs, otherwise -1
 *
 ************************************************************************/
word *
PSX_WRITE ( word *para[] )
{
        word stream, amount, result, ret;
        char *buff;

        stream = *para[0];
        buff   = (byte *) para[1];
        amount = *para[2];

        while ( amount > 0 )
        {
                result = write (stream, buff, amount);
                if ( result <= 0 )
                {
                        ret = -1;
                        return &ret;
                }
                amount -= result;
                buff = &(buff[result]);
        }
        ret = 0;
        return &ret;
}
```

Working with farms:

```
/* gen.c
 *
 * NUMBER GENERATOR
 *
 * - used as a front-end for a farm of worker tasks to test the behaviour
 *   of the load balancer.
 * - transfers random initialized data arrays to the worker processes
 * - to be linked with a module whic defines full_read() and full_write
 *
 */

#include <helios.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#include <syslib.h>
#include <posix.h>
#include <lb.h>
#include <nonansi.h>

#define DIM     2
                                               /* The data packet used for */
typedef struct datapkt {                /* the data transfer:        */
        LB_HEADER       header;                /* Used by the load balancer*/
        double          data[DIM][DIM];        /* The data: an array       */
        int             index;                 /* For identification       */
} datapkt;

void    reader ( int count );

static datapkt  r_pkt;                         /* For receiving data       */
static datapkt  d_pkt;                         /* For sending data         */

static debug = 0;                              /* Default: no debugging    */

Semaphore wait_term;                           /* Used for synchronization */
                                               /* at termination point     */
```

```
/*****************************************************************************
 * THE WRITER , SENDING ARRAYS TO THE WORKER PROCESSES
 *****************************************************************************/
int
main ( int argc , char *argv[] )
{
        int count, i, j;

        if ( argc < 2 || argc > 3 )             /* Check the arguments       */
        {                                       /* -d : debugging            */
                printf ("Usage: %s <number of arrays to be calculated> [-d]\n",
                        argv[0]);
                return 1;
        }

        count = atoi ( argv[1] );
        if ( argc > 2 && argv[2][1] == 'd' )
                debug = 1;

        for ( i = 0 ; i < DIM ; i++ )           /* Initialize the array      */
                for ( j = 0 ; j < DIM ; j++ )
                        d_pkt.data [i][j] = 1.234;

        InitSemaphore ( &wait_term, 1 ); /* For correct termination  */

        Fork ( 5000, &reader, 4, count );       /* Create the receiver pro- */
                                                /* cess.           */

        d_pkt.header.size    = sizeof (datapkt) - sizeof (LB_HEADER);
        d_pkt.header.control = 0;

        for ( i = 0 ; i < count ; i++ )         /* Transfer the sequence    */
        {                                       /* of arrays to the lb      */
                d_pkt.index = i;
                                                /* Wait, until write succeeds*/
                full_write ( 5, (char *) &d_pkt, sizeof (datapkt) );
                if ( debug )
                {
                        printf ("gen: Written array no: %d\n", i );
                        fflush (stdout);
                }
        }
                                                /* Prepare and transfer the */
                                                /* termination message      */
        d_pkt.header.size = 0;
        d_pkt.header.control = LB_MASTER + Fn_Terminate;
        full_write ( 5, (char *) &d_pkt.header, sizeof (LB_HEADER) );
```

```
                                                        /* Wait for the receiver    */
                                                        /* process to terminate.    */
        Wait ( &wait_term );
        printf ("gen: finished\n");
        fflush (stdout);
        exit (0);
}


/*******************************************************************************
 * RECEIVE RESULTS FROM FARM WORKER COMPONENTS
 ******************************************************************************/
void
reader ( int count )
{
        int     i, j;

        Wait ( &wait_term );                            /* Note that the receiver   */
                                                        /* is active                */
        for ( i = 1 ; i <= count ; i++)                 /* Expect the exact number  */
        {                                               /* of packets               */
                full_read (4, (char *) &r_pkt, sizeof (datapkt) );

                if ( debug )
                {
                        printf ("gen: Received array no: %d\n", r_pkt.index);
                        fflush (stdout);
                }
        }
        Signal ( &wait_term );                          /* Allow the sender to      */
                                                        /* terminate the task       */
}
<eof>
```

```
/* work.c
 *
 * FARM WORKER-TASK
 *
 * - this is one element in a processor farm which is connected via the
 *   the load balancer to the master task.
 * - reads arrays of double, performs some calculations and transfers them
 *   back
 *
 */

#include <helios.h>
#include <stdio.h>
#include <stdlib.h>
#include <syslib.h>
#include <math.h>
#include <posix.h>
#include <lb.h>

#define DIM     2

typedef struct datapkt {
        LB_HEADER       header;
        double          data [DIM][DIM];
        int             index;
} datapkt;

static struct datapkt   d_pkt;

int
main ( )
{
        int     i, j, repeat;
        double  result;

        for ( ;; )
        {                                       /* Expect the data packet   */
                full_read ( 0, (char *) &d_pkt, sizeof (datapkt) );
                                                /* We have received a ter-  */
                                                /* minate request ?         */
                if ( ( d_pkt.header.control & LB_FN ) == Fn_Terminate )
                        break;

                                                /* This is a (dummy) calcu- */
                                                /* lation part.             */
                for ( repeat = 0 ; repeat < 10000 / DIM ; repeat++ )
                        for ( i = 0 ; i < DIM ; i++ )
                                for ( j = 0 ; j < DIM ; j++ )
                                        result = sin (d_pkt.data [i][j])/0.234;
```

```
                                              /* Prepare the reply          */
                d_pkt.header.size = sizeof (datapkt) - sizeof (LB_HEADER);

                                              /* Write the result of the   */
                                              /* calculation back          */
                full_write ( 1, (char *) &d_pkt, sizeof (datapkt) );
        }
        exit (0);
}
<eof>
```

The CDL-script used :

```
#! /helios/bin/cdl

component lb {
        memory      200000;
        code        /helios/bin/lb;
}

component gen {
        memory      80000;
}

component work {
        memory      30000;
}

gen $2 -d [$1]||| work
```

Diagnostics from the Task Force Manager

```
... start ...

/Cluster/tfm : create request received
/Cluster/tfm : calling create_tf
/Cluster/tfm : create_tf, building cdl object for cdl.tf.1
/Cluster/tfm : create_tf, creating mapping structure
/Cluster/tfm : create_tf, performing mapping
/Cluster/tfm : Overall Mapping quality for Phase1 1e
/Cluster/tfm : mapping done
/Cluster/tfm : allocating mapped resources
/Cluster/tfm : executing components
/Cluster/tfm : executed /Cluster/PC/helios/c/cdl/demos/s_ on /Cluster/01/tasks
/Cluster/tfm : executed /Cluster/PC/helios/c/cdl/demos/m_ on /Cluster/02/tasks
/Cluster/tfm : create_tf, installing monitor processes
/Cluster/tfm : creating monitors for cdl.tf.1.1
/Cluster/tfm : create_tf, creating connecting streams
/Cluster/tfm : task /Cluster/01/tasks/s_.6, calling GetProgramInfo
/Cluster/tfm : task /Cluster/02/tasks/m_.5, calling GetProgramInfo
/Cluster/tfm : created cstream object /Cluster/01/pipe/S1.1
/Cluster/tfm : created cstream object /Cluster/02/pipe/S0.1
/Cluster/tfm : create_tf, done
/Cluster/tfm : create done
/Cluster/tfm : open request received
/Cluster/tfm : successfull open
/Cluster/tfm : stream_server for tf cdl.tf.1.1
/Cluster/tfm : stream_server for cdl.tf.1.1, received environment
/Cluster/tfm : stream_server for cdl.tf.1.1, ProgramInfo, port c0560019

... execution ...

... termination ...

/Cluster/tfm : task /Cluster/01/tasks/s_.6 terminated with 0
/Cluster/tfm : tf cdl.tf.1.1, component terminated, 1 left
/Cluster/tfm : task /Cluster/02/tasks/m_.5 terminated with 100
/Cluster/tfm : tf cdl.tf.1.1, component terminated, 0 left
/Cluster/tfm : deleting program /Cluster/01/tasks/s_.6
/Cluster/tfm : deleting program /Cluster/02/tasks/m_.5
/Cluster/tfm : tf cdl.tf.1.1, sending terminate to c0560019
/Cluster/tfm : tf cdl.tf.1.1, waiting for 1 Close requests
/Cluster/tfm : stream_server, close request
/Cluster/tfm : stream_server for cdl.tf.1.1 finishing
/Cluster/tfm : deleting tf cdl.tf.1.1
/Cluster/tfm : delete_tf for cdl.tf.1.1, removing task force from directory
/Cluster/tfm : delete_tf, calling terminate
/Cluster/tfm : delete_tf, freeing task list
/Cluster/tfm : delete_tf, releasing mapped resources
```

```
/Cluster/tfm : delete_tf, free'ing connecting streams
/Cluster/tfm : Deleting pipe /Cluster/02/pipe/S0.1
/Cluster/tfm : Deleting pipe /Cluster/01/pipe/S1.1
/Cluster/tfm : delete_tf, free'ing mapping structure
/Cluster/tfm : delete_tf, free'ing cdl object
/Cluster/tfm : delete_tf, free'ing TASK_FORCE structure
/Cluster/tfm : delete_tf, done
/Cluster/tfm : Mem free is 24780, heap is 31964

... terminated ...
```