The Hub:
Moving Data Between Minicomputers Using an Interconnection Language

Thomas R. Miller

Department of Computer Science
University of Waterloo

Research Report CS-77-12

April 1977

The Hub:
Moving Data Between Minicomputers
Using an Interconnection Language

Thomas R. Miller

A thesis
presented to the University of Waterloo
in partial fulfillment of the requirements
for the degree of
Master of Mathematics
in
the Department of Computer Science

Waterloo, Ontario, 1977

Abstract

The Hub is a program designed to facilitate data transfers between minimally configured minicomputers arranged in a star network around a central machine, on which the Hub program runs. Peripheral devices may also be shared among the machines this way. A module interconnection language is defined for creating data transfer programs. Users enter programs in this language and the Hub executes the programs. Guarantees are made about the reliability of the Hub as a whole. The program is written in a high-level language and is highly portable.

This work would not have been possible without the support and direction provided by Prof. Michael Malcolm. The facilities and environment provided by the University of Waterloo Mathematics Faculty Computing Facility were also extremely helpful. Many people have contributed to the growth of the Hub. Among these are: Bert Bonkowski, John Corman, Morven Gentleman, Gary Sager, Gary Stafford and Fred Young.

Table of Contents

1. Introduction

In an environment with minimally configured minicomputers and one or more larger computer systems, it is desirable to be able to transfer data between machines. For example, one might want to load programs into one machine from another. The ability to share available peripheral devices (such as disc or tape drives) enhances the utility of such devices. Minicomputer files may be backed up using the facilities of a larger system. Acquiring a machine and adding it to the environment becomes less expensive since there is no need for the new machine to have expensive peripheral devices of its own.

This thesis is a description of the design and implementation of such a system in the Real-Time Minicomputer Laboratory at the University of Waterloo. For a description of the Minicomputer Laboratory, see Malcolm and Sager (1976).

One way to provide these services is to write software for each machine that will enable it to communicate with all of the other machines to which it is connected. As each machine has its own instruction set, specifications and idiosyncracies, this can involve a large amount of machine-specific programming. Each time a new machine is added to the configuration, the job must be repeated for that machine. Whenever a new connection between machines is made, more programming may be required at each end of the connection. In addition, many interfaces and cables would be needed for such a completely connected set of machines.

A better way to handle this problem is to designate one machine as the **Hub** of a star network, and place the responsibility for data transfers and inter-machine communication on that machine. This requires fewer interfaces for the peripheral machines and fewer cables between machines, as each periphal machine communicates only with the Hub machine. Given a reasonably static environment, this proposal would be sufficient. However, hardware failures in the Hub machine or a change in the desired use of the network which requires a different machine to take over some of the Hub duties make one further step necessary, namely to implement the Hub to be portable. This means that although only one machine may play the role of the Hub, the particular machine which must do so need not be chosen beforehand. Upgrading the Hub to run on newer hardware is also made easier by this.

The term "star network" is used because the central Hub machine has access to all of the machines in the network via communication lines. Most large multi-user computer systems cannot easily be adapted for such use, because they require the remote device (terminal or computer) to initiate communications in some prescribed manner. However, a minicomputer requires that a program

be loaded into it to follow whatever pattern is required for communications. Also, most minicomputers have ROM loaders available. The Hub machine must be able to communicate with these ROM loaders in order to load the minicomputers. In general, the Hub must be able to use whatever communications protocol the peripheral machines are using. From the point of view of a peripheral machine, the network exists only when use is being made of it. The star configuration allows data to be transferred while keeping most of the machines free for other use.

In the Hub network, data files can be prepared on one computer and then transferred from that machine to another. For example, cross software may be used on a large computer to prepare executable code for use by one of the minicomputers. Such cross software includes compilers for high-level languages, assemblers and linking loaders. Executable data can then be loaded into the target machine via the Hub.

The Hub is currently running on a Texas Instruments 990/10 minicomputer. Applications include loading a Microdata 1600/30 and a TI 990/4 from files kept on the Hub machine's disc. As new machines are brought into the Minicomputer lab, they can be attached to the Hub and programs run on them in very little time.

The Hub implementor cannot know in advance what tasks the Hub will be used to accomplish. To achieve the desired flexibility, a module interconnection language was designed for the Hub. Tasks to be carried out by the Hub are expressed as programs in that language. Basically, this module interconnection language provides a means for specifying the manner in which a set of predefined software modules is to be connected, where connections represent data flow between modules. The Hub need only contain a "stock" set of module definitions and provide the user with a means of entering and executing a program which specifies how to interconnect some subset of the modules. A set of predefined programs (macros) is provided for common uses of the Hub.

Since little is known about the implementation and use of module interconnection languages, the Hub also provides a setting in which to explore the usefulness, strengths and weaknesses of this approach.

2. Thoth: Background

This chapter gives some essential background information regarding the operating system under which the Hub runs. The choice of operating system is very important, especially in light of the portability goals mentioned in the introduction. The operating system chosen has recently been developed at the University of Waterloo.

The Hub runs under the real time executive Thoth (Cheriton, et. al., 1977). Braga (1976) describes the base language for Thoth and the Hub. This language is a descendant of the language "B" (Johnson and Kernighan 1973), and was explicitly designed for portability. Compilers are currently available for the Data General NOVA 2, the Texas Instruments 990 and the Honeywell 6000 series machines. The portability of the Hub depends upon the portability of Thoth.

Like B, the Thoth base language is stack-oriented, permitting reentrant and recursive programming. Thoth provides a machine abstraction designed to cover a wide range of available and future machines. Restrictions defining the class of machines on which Thoth may be implemented are described in Cheriton, et. al. (1977).

Thoth creates an abstract machine environment which provides such concepts as processes and a clock. Thoth also provides dynamic memory allocation and recovery, which is important to the Hub. Any Thoth program, such as the Hub, consists of some set of concurrent interacting processes. Thoth itself is composed of many such processes, and the user program it supports may be quite complex.

A Thoth process is an execution instance of a function. Functions called as subroutines use the same stack as the calling function, while a function invoked as a process is given its own stack. The stack for a process is allocated when the process is created. Each process also has a unique one-word **process id** assigned to it when the process is created. This provides a mechanism for referring to specific processes. More than one process may be executing the same function.

One attribute of a process is its "status", or state. This can be one of **embryonic, ready, asleep, receive-blocked, send-blocked,** or **awaiting-interrupt.** When a process is created, it is initially in the embryonic state. A process is not considered for execution unless its status is ready. An embryonic process can be made ready only by a call to the Thoth function .Ready.

Another important attribute of a process is its **priority.** Thoth provides a set of priority levels, one of which must be specified when a process is created. There is also an external variable which can be used to set the priority level at which the root of the user's tree of processes executes. Priority levels can

be used to achieve **relative indivisibility** between processes. A higher priority process cannot be interrupted by a lower priority process. A process of a given priority level will not run as long as there are ready processes at higher priority levels. The use of priority levels in the Hub will be discussed in greater detail later.

Thoth also provides a construct called a **team.** A team is a set of processes which share memory. The root process of a team is given a specific amount of memory when created. A team "owns" its free-list of memory; all allocations must come from that free list. No process from another team may allocate memory from the first team's free-list. The stack for each process in a team comes from the team's free-list. When a team dies (or terminates), its entire block of memory is freed, enabling complete recovery of its resources. The amount of memory owned by a team cannot be increased after the team is created. While there may be free memory elsewhere, attempts by a team to allocate more memory than it owns will fail. This makes accurate predetermination of a team's memory requirements crucial.

Interaction between processes in Thoth occurs mainly by use of the communications primitives .Send and .Receive. These functions transfer one word of data between the sending process and the receiving process. They are synchronized in the sense that the receiving (sending) process will block until the sending (receiving) process has executed the corresponding .Send (.Receive).

Of the guarantees that Thoth makes to the user, three are very important to the Hub:

1) A process which attempts to receive data from a dead or non-existent process will die without executing further.
2) When a Thoth process dies, all of its descendants are automatically killed.
3) When a Thoth team dies, all of its memory resources are recovered.

3. Team Interconnection Language

This chapter describes the form of module interconnection language provided by the Hub for creating data transfer programs.

A major design criterion for the Hub is flexibility. There are a few obvious jobs for it to do in the Minicomputer Laboratory, but beyond these there is no set structure of capabilities to be frozen into the design. New applications can be easily implemented since the Hub provides a high level programming language.

The general requirement of Hub applications, data movement, is met by providing a set of "building block" software modules which the user connects to achieve given applications. Each module performs some operation on the data which flows through it. The programming language implemented by the Hub is thus a form of module interconnection language. It is called the Team Interconnection Language (TIL) because each module is a Thoth team.

Previous work on module interconnection languages has stressed their use as tools for organizing large programs into smaller, more manageable modules. DeRemer and Kron (1975), for example, state the objectives of a module interconnection language as:

1) to aid in project management, in facilitating group programming efforts,
2) providing the ability to prove correctness of the individual modules, and
3) deriving the correctness of the overall program from the correctness of the modules and the rules for their combination.

In DeRemer and Kron's paper, interconnections serve to define modular breakdowns of tasks into subtasks. TIL is somewhat different, although the second and third points above are relevant. In TIL, interconnections are used to define data communication paths between concurrent processes.

TIL can be kept more "user proof" than a general-purpose programming language, and is more flexible than a set of completely predefined application programs. The TIL approach also facilitates portability. Most modules contain nothing dependent upon a particular hardware environment. Machine-specific modules can be easily added when they are necessary.

High levels of reliability and testability can be achieved with such a modular design. Edwards (1975), in comparing software and hardware design principles, confirms this. Each team performs a small well-defined task, and can be tested quite rigorously before it is used in conjunction with other

teams. Resource requirements for individual teams are kept independent of the way in which the modules are combined.

The objects that are manipulated in TIL are **teams** and **connections.** Teams will be described in detail later. Connections are unidirectional data paths between teams. A team may have up to ten input and ten output connections. A simple conceptual example of a team would be the "Tee", illustrated in Figure 1, which takes data from a single input and copies it to two outputs.

The Hub's main purpose is to transfer data. Thoth provides the single-word message passing functions .Send and .Receive. The Hub contains analogous built-in functions (.Send_buf and .Receive_buf) which pass buffers of data between processes. The results of timing studies of these two methods of data transfer for three machines are shown in Table 1. The times in Table 1 represent the time used to transfer one unit (word or buffer) of data, and include all of Thoth's overhead costs. These tests essentially consist of two teams and a single connection. One team repeatedly sends data and the other repeatedly receives data. No device i/o is done at either end of the transfer, so the baud rates in Table 1 are upper bounds.

Table 1: Throughput for a simple data transfer

bd : baud (10 bits per byte, 2 bytes per word)

Method	TI 990/10	TI 990/4	DG NOVA 2
Word-passing	615 μs 32520 bd	779 μs 25674 bd	250 μs 80000 bd
Buffer-passing (8 word buffer)	813 μs 196802 bd	1004 μs 159363 bd	468 μs 341880 bd
Buffer-passing (16 word buffer)	927 μs 345200 bd	1143 μs 279965 bd	590 μs 542372 bd

The buffer passing functions operate on fixed length buffers. One buffer each is allocated by the sending and receiving processes. Data is physically copied from the sender's buffer into the receiver's buffer. The count of the number of words containing "real" data is kept in the first word of the buffer. This is a simple and easily programmed method. Other methods require more machinery to be provided. One alternate method is to pass the address of the data buffer between teams. For this to work, some sort of buffer allocation and recovery scheme must be provided. In addition, it is more difficult to guarantee data protection. Requiring each process to allocate its own buffer (instead of shar-

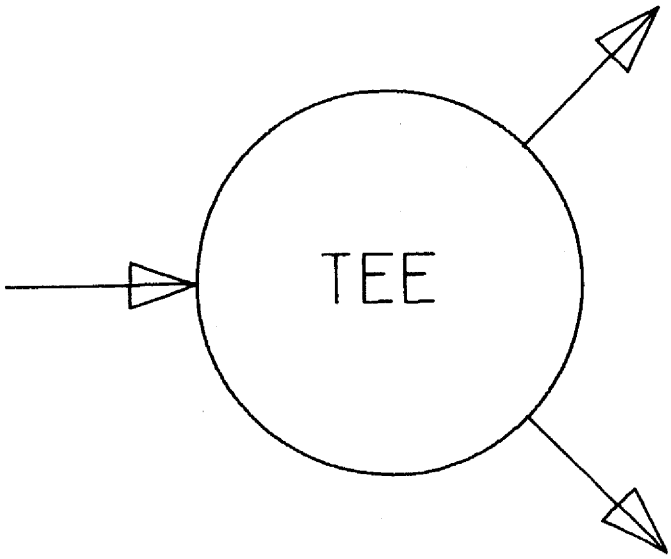

Figure 1. The "Tee", a simple team

ing buffers) makes the data transfer more reliable. One process cannot accidentally destroy the data that it has sent to another process; a buffer is free for re-use as soon as it has been sent. The Hub buffer-passing functions are shown below:

```
.Send_buf( receiver_id, buf )

\    buffer-passing counterpart of .Send
\    major difference:  does a .Copy from buf to BUFFER[receiver]

  {
   extrn .Active, .Pid_base, .Idmask;
   auto i, rec;

   disable;
   rec = .Pid_base[receiver_id&.Idmask];
   if ( ID[rec] != receiver_id )
     {
      enable;
      return;
     }
   if ( STATUS[rec] == REC_BLOCKED
        && BLOCKED_ON[rec] == .Active )
     {
      .Copy( BUFFER[rec], buf, BUFFER_LEN[buf] );

      .Add_ready( rec );
      return;
     }
   BUFFER[.Active] = buf;
   BLOCKED_ON[.Active] = rec;
   STATUS[.Active] = SEND_BLOCKED;

   i = SENDQ_TAIL[rec];
   BLOCK_FWD[.Active] = i;
   BLOCK_BACK[.Active] = BLOCK_BACK[i];
   SENDQ_TAIL[rec] = .Active;
   BLOCK_BACK[i] = .Active;

   .Block();
  }
```

```
.Receive_buf( sender_id, buf )

\    buffer-passing counterpart of .Receive
\    does a .Copy from BUFFER[sender] to buf
\    and returns the number of data words in
\    buf
\    commits .Suicide if the sender does not exist

  {
   extrn .Active, .Pid_base, .Idmask;
   auto i, sen;

   disable;
   sen = .Pid_base[sender_id&.Idmask];
   if ( ID[sen] != sender_id )  .Suicide();

   if ( STATUS[sen] == SEND_BLKED
        && BLOCKED_ON[sen] == .Active )
     {
      BLOCK_BACK[BLOCK_FWD[sen]]  =  BLOCK_BACK[sen];
      BLOCK_FWD[BLOCK_BACK[sen]]  =  BLOCK_FWD[sen];

      .Copy( buf, BUFFER[sen], BUFFER_LEN[BUFFER[sen]] );

      .Add_ready( sen );
      enable;
      return( BUFFER_LEN[buf] );
     }
   BUFFER[.Active] = buf;
   BLOCKED_ON[.Active] = sen;
   STATUS[.Active] = REC_BLKED;

   .Block();
    return( BUFFER_LEN[buf] );
  }
```

To write a TIL program, the user must first know what types of teams are available for use, what the function of each team is, and how many input and output connections each team may make. Each distinct team is represented in the Hub by its **prototype,** which is the code executed by the team. Each prototype has a set of **attributes.** These are name, start address, the amount of memory required to run the team, and the amount of stack needed by the root process of the team.

A TIL program consists of two parts: the **team naming** part and the **connections** part. In the team naming part, the user specifies how many teams are wanted in the program. This is done with a series of lines of the form:

<prototype name> <list of team names>

which bind the names in <list of team names> to represent teams of type <prototype name>. A single argument may optionally be supplied with each name in the list. Specification of the argument can be left until execution time by using empty parentheses. This argument will be passed to the team when it is readied. Simple examples of team naming lines would be:

```
File_reader   X( $tty0 ), Y( file3 )
Micro_load    Z
```

In the first example, "File_reader" is the name of a Hub prototype, "X" and "Y" are names for the two instances of this prototype, and the strings "$tty0" and "file3" are to be passed as arguments to the teams X and Y, respectively. In the second example, no argument is specified for the Micro_load team. Every team named in the team naming part will be included in the TIL program.

Once the teams have been named, the user must specify how connections are to be made between them. This is the connections part of the TIL program. Any team which is used in the connections part of a TIL program must have been named in the team naming part. A team can have zero or more of both **mandatory** and **optional** connection ports. All of the mandatory ports must be used in the connections part, while any number of the optional ports may be used.

Only one connection may be specified per line. The output port must be first. Connection lines have the form:

<output team> <port> <input team> <port>

where the <output team> and <input team> are team names defined in the team naming part and the <port>s determine which ports are being connected. Each <port> must be an integer between 0 and 9, inclusive. These may be separated by one or more blanks and dashes. An example of a connection line is

```
X - 1   Y - 2
```

where output port 1 of team X is to be connected to input port 2 of team Y.

A simple TIL program to read the file "data/test" and print it on the device "$tty3" might be written as follows.

```
file_reader  a( data/test )
tty_out  b( $tty3 )

a - 0    b - 0
```

Here it is assumed that file_reader teams have one mandatory output and no mandatory inputs, and tty_out teams have one mandatory input and no mandatory outputs.

One way to visualize a TIL program is as a directed graph, where the teams are nodes and the connections are edges. The example just given is shown in Figure 2. A given application can be formulated in terms of such a diagram and then translated into a TIL program.

For a formal definition of the Team Interconnection Language, see Appendix 4.

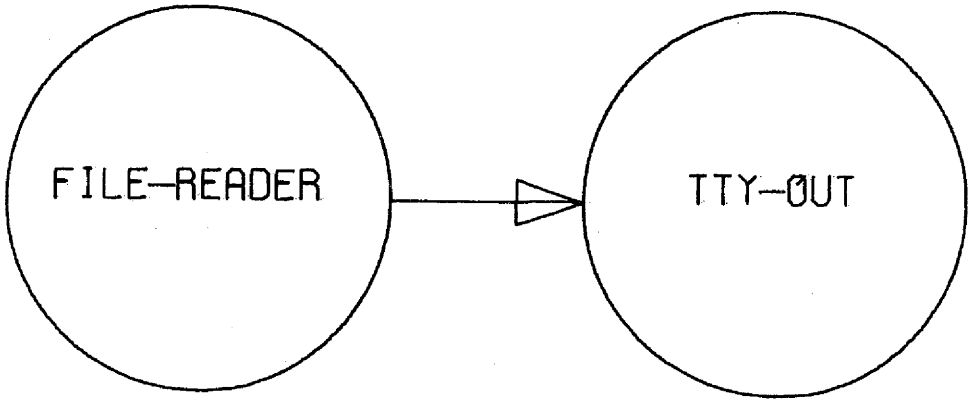

Figure 2. Example TIL graph

4. Teams and Prototypes

A TIL team is an execution instance of a prototype. A prototype is the code executed by a Thoth team. Different prototypes may share code for common functions.

In general, a team consists of several processes. In practice there are many teams which have only one process. TIL team processes are of two types: **port** processes and **internal** processes.

Internal processes are those which communicate only with processes within the same team. They may perform some computation or synchronization.

Port processes are responsible for communicating with port processes of other teams. The Hub buffer-passing primitives .Send_buf and .Receive_buf are used to accomplish this.

The most complex process in most teams is the root process. The root may create other processes in the team. The root must make sure that all resources required by the team are successfully acquired. Each team is given, as the team's memory resources, the amount of memory needed for the team to perform its task. This is a predefined quantity and must include the space needed to create other processes, open files, allocate memory for buffers, etc. Memory requirements also include the space needed for the stacks used by all processes in the team. Determining the number of words of stack needed by a given process is difficult. In general, determining an upper bound is the best that can be done. The amount of stack needed by a single function, on the other hand, can be determined exactly. This area of the process's stack (called the function's **stack frame**) contains space for four kinds of quantities: arguments expected by the function, variables local to the function, vectors declared as part of the local variables, and temporary variables used by the compiler in evaluating expressions.

Each process has a **call graph,** where nodes represent functions and edges are directed from a function to all of the functions it calls. When there is no recursion present, this graph is actually a tree. If then each node carries a weight equal to the amount of stack needed by the function it represents, then the amount of stack needed by the process at the root of the tree corresponds to the longest path from the root to a leaf.

Recursion and indirect function calls complicate this analysis, so that it is only feasible to bound the amount of stack needed by a process. An interactive program is being developed which uses information generated by the compiler to help a user generate stack bounds. There is still much work to be done in this area.

5. Reliability

Reliability is an important attribute of the Hub. The Hub should be able to run until new prototypes need to be added to it, or the machine is stopped for maintenance or repairs. The Hub is reliable because it will not attempt anything which might cause it to fail destructively. When there is not enough memory available to perform some task, that task is not attempted. The commands available are well-defined and the Hub has been designed so that none of them can cause a fatal error.

Memory integrity must be maintained for the Hub to be reliable. This involves three problem areas. Bugs in the Hub program can cause destruction of code or data. By keeping the Hub code simple and easily understood, bugs are minimized. The second source of trouble is the possibility of stack overflow. These two problems must be solved through a combination of careful design and analysis of memory and stack requirements. The third aspect of memory integrity is protection of the data that the Hub transfers. This protection is provided by a combination of the Thoth team concept, the design of the Hub buffer passing functions and correct coding in the teams themselves.

Memory recovery is crucial to the Hub. If even small amounts are not recovered from time to time, the Hub will eventually deadlock due to a lack of free memory. The major users of memory are TIL programs, although most other Hub functions build small data structures during execution. Care is taken to ensure that any memory allocated in a Hub function is freed (if not needed permanently) before the function returns.

Recovery of memory used by TIL programs presents a slightly different problem. A TIL program is composed of several Thoth teams, each of whose allocated memory block is recovered when its root process dies. In order to guarantee complete recovery of the resources used by any TIL program, all of the the teams in the TIL program must die. We say that a TIL program has "terminated" when all of its teams have died. The Hub performs a test (described later) and admits as legal only those TIL programs which will terminate.

The input connections of a node are all those nodes which have edges directed to it. The TIL graph corresponding to any TIL program consists of two types of nodes: source nodes and regular nodes. Source nodes have no input connections, while regular nodes have input connections but might not have output connections. Every TIL graph contains at least one source node. They occur wherever data is taken directly from a file or device, and are used for the "originating" points of data transfers.

There are two restrictions. First, each node must be designed so that when all of its input connections have died, it will eventually die also. This implies that source nodes must always eventually die "by their own hand", as they have no input connections. Second, the TIL graph must contain no cycles. With these restrictions, we are led to the following theorem.

Termination theorem:

Given that:
1) Source nodes eventually die of their own accord.
2) A node dies when all of its input connections die.
3) The TIL graph contains no cycles.

Then:
All of the nodes in the TIL graph will eventually die.

Proof:

Let N be the set of all nodes in a TIL graph corresponding to our premises. We can define subsets of N called **levels** as follows: all source nodes of N are in level 0; the level of any node X which is not a source node is one plus the maximum of the levels of the input connections of X. For example, a node connected only to source nodes would be in level 1. In an acyclic graph, the notion of level is well defined.

We proceed by induction on the levels of N. A level is said to die if all of the nodes in the level die. Level 0 is assumed to eventually die, as it is composed only of source nodes. Now assume that all levels up to and including level i are dead, and consider level i+1.

The nodes in level i+1 will die eventually if all of their input connections die. By the definition of level, each input connection must lie in some level j where $0<=j<=i$. Since all nodes in those levels are assumed to have died, all input connections to the nodes in level i+1 must be dead. This implies that all nodes in level i+1 will eventually die. Thus, by induction, all nodes in N must eventually die. This proves the theorem.

A TIL graph may consist of two or more disconnected subgraphs. It should be noted that the proof also holds for such graphs.

Properties needed by individual teams for the above theorem to hold must be designed into the teams. Satisfying the first assumption is not difficult. When a source team reaches an end of file or

other termination condition, it may execute the statement

```
.Destroy( ID[TEAM[.Active]] );
```

which kills the team root process, and thus the team.

Satisfying the second assumption is more difficult, and proves impossible for some applications. It is clear that if a team becomes permanently blocked and as a result never attempts to receive any data after its input connections die, it will not necessarily die. Thus, a team must be designed so that at any point in time, all of its input ports will eventually attempt to receive data. This must be shown for each team, based on the design of the team and the assumption that output ports do not block indefinitely. This assumption can be shown to hold if the input ports of all teams are guaranteed not to block indefinitely, since the output ports are connected to such input ports and thus will not block indefinitely.

Even if the inputs are guaranteed not to block indefinitely, death of the input connections still implies only the death of the corresponding input port processes. In order that the death of the team follow, we might have the team root acting as an input port process, or receiving data from an input port process.

A simple example of a team which cannot be designed so that its input port processes must always receive again is a team which merges two input streams by alternating, one buffer at a time, between the two inputs. One input must remain blocked until the other has passed its data through. This may end up as a permanent block, if a situation develops where one input port is waiting for data but the only data available is destined for the other input port. Thus we cannot guarantee that at any point in time, both input port processes will eventually attempt to receive data again. Such teams should not be used in the Hub unless there is no other way for them to be designed. However, since the Hub provides the ability to kill (or terminate) TIL programs, termination can still be achieved.

The only property that a TIL graph as a whole must have is that there be no cycles present. This must be checked when each TIL program is submitted for execution. The method used by the Hub is somewhat novel: the TIL graph is simulated by a set of nodes configured to have the same graph structure as the TIL graph. Each node executes a .Receive_any for each of its input connections, then sends one message to each of its output connections, and dies. Following is the simulation node function.

```
Node( in_degree, out_degree, out_neighbors )

\   simulation node:  used to discover whether
\   there are cycles in a TIL graph

  {
    auto i;

    while( in_degree > 0 )
      {
        .Receive_any();
        --in_degree;
      }
    for( i=0; i<out_degree; ++i )
        .Send( out_neighbors[i], 0 );
  }
```

The node processes in the simulated TIL graph run at a higher priority than the process which is running the simulation. Once all nodes in the graph have been readied, control should not return to the process running the simulation until either all of the nodes are dead or some set of them is deadlocked in **receive-blocked** states. At this point, the graph can be examined to determine whether any nodes still exist. If no nodes exist, then there were no cycles in the TIL graph.

Source nodes, whose in_degree is zero, send messages and die. All other nodes must first receive messages from nodes on lower levels and then send messages to nodes on higher levels. If there are no cycles in the TIL graph, then, following the argument given in the proof of the Termination theorem, all of the simulation nodes must die. However, if there are cycles in the TIL graph it is not possible to partition the graph into levels. In this case, the nodes in the cycle will block permanently, waiting to receive from each other. Any other nodes waiting to receive from nodes in the cycle will thus also be blocked permanently.

It is interesting to note that this method is linear in the number of vertices and edges in the TIL graph. Each node must be created and readied, and then for each edge (connection), a message must be sent and received. For a discussion of other graph algorithms, and in particular for another linear algorithm, see Tarjan (1972).

6. Design and Implementation

The Hub implementation makes use of Thoth priority levels in the following way. There are four classes of processes in the Hub: user interface processes, termination simulations, initiator processes, and TIL processes. The function Main initializes the Hub and becomes the default user interface process (called the **console**). User interface processes create termination simulations and initiators; initiators create TIL programs. User interface processes run at level 6 (note that higher priority level numbers indicate lower priorities). The termination simulations run at level 5, as do the initiator processes. TIL team roots run at level 7, although the structure of some teams may require processes which run at priority levels other than 7. Figure 3 provides a schematic illustration of these relationships.

The function Main goes through two stages. First the table containing the attributes of the available TIL teams is built. Then the console begins execution.

Each user interface process allows Hub commands to be entered and executed. User interface processes communicate with devices, usually terminals or the peripheral machines. The console uses the main terminal of the Hub machine. There are two user input modes. The first is command input mode, where a question mark is used for a prompt. The second is entry of TIL programs, where a colon is used as a prompt. Following is a summary of Hub commands.

Command	Description
delete	delete a macro from the Hub.
kill	destroy the initiator process of a TIL program. This kills the TIL program also.
list	produces a list of active initiator processes. This corresponds to TIL programs that are still running.
macros	produce a list of macros.
q	terminate the user interface process. At default console, also terminates Hub program.
teams	produce a list of teams.
terminal	spawn a new user interface process. Prompts for device.
til	enter and run a TIL program.
<macro name>	run the TIL macro <macro name>.

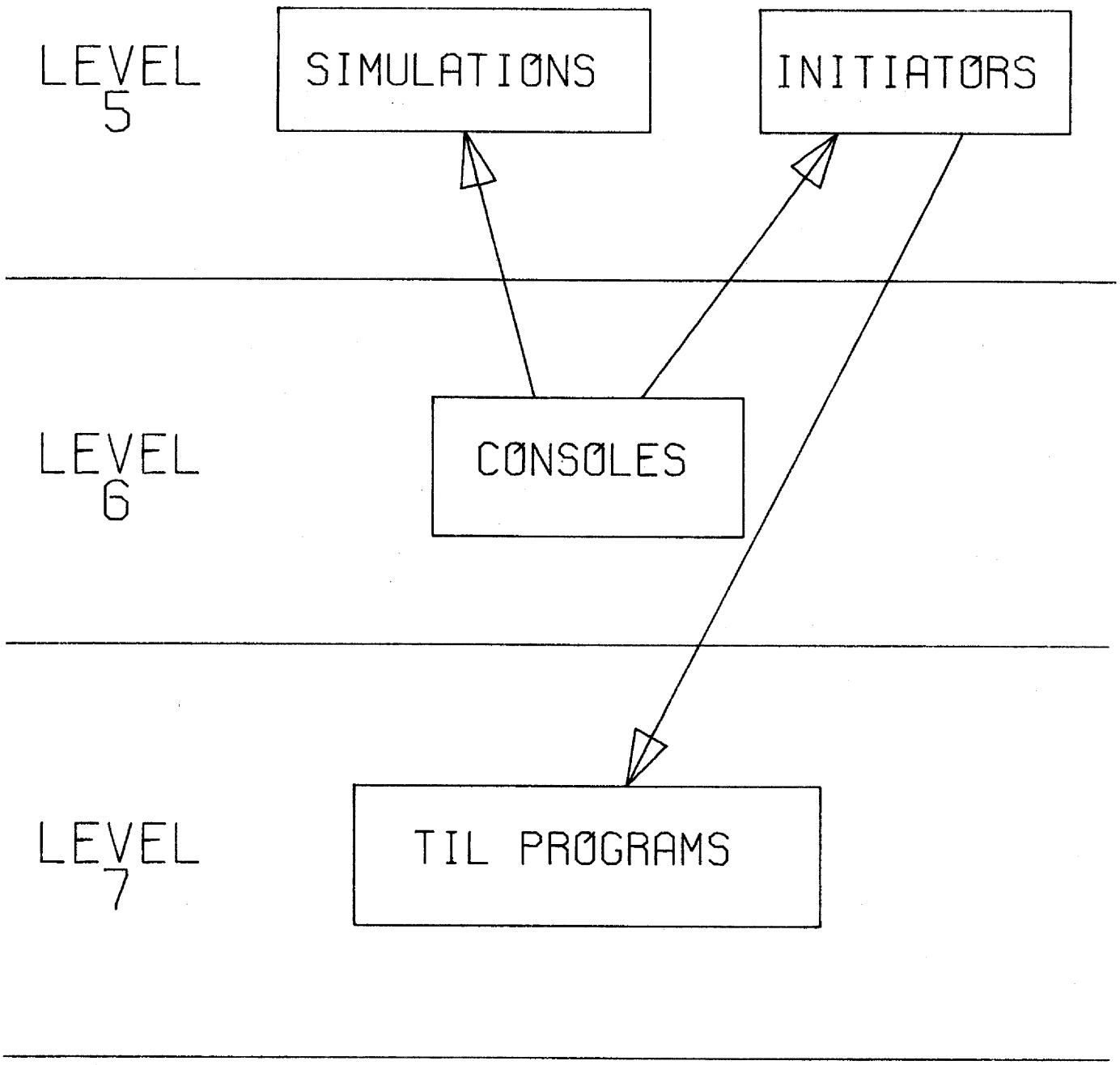

Figure 3. Use of priority levels

Before proceeding, a short definition of "TIL macro" is in order. TIL macros grew out of the realization that no matter how dynamic and/or developmental is the environment of the Hub, there will always be some applications which will be made over and over. A macro is the text for a TIL program which is kept in a file for later use. There is a TIL team for the purpose of entering the text; thus, a macro can be created for defining macros. Each macro has a unique name associated with it. When a macro name is given at command input level, the input for TIL program definition is then taken from the macro file instead of the user's terminal.

Initiator processes are responsible for running TIL programs. Each initiator stays alive as long as its corresponding TIL program executes. This allows the user to destroy a TIL program by destroying its initiator process. The "kill" facility admits the possibility of TIL programs composed of teams which do not meet the requirements of the termination theorem.

To enter and run a TIL program, the user types the command "til" at a Hub console. The console prompts the user for a series of lines which are taken as the team naming part of the TIL program. A null line terminates the team naming part, and the console then prompts for a series of lines which form the connections part of the TIL program. A null line or the keyword "wait" terminates the connections part. At this point, any arguments left unspecified are prompted for. This is most useful for TIL macros, since certain arguments for the teams in the macro may be different each time the macro is used. Errors made at any point cause the console to print a brief error message and return to command input mode.

If there were no errors detected in the TIL program input, the termination simulation previously mentioned is performed. If this fails, the TIL program is not run. Otherwise, an initiator process is created for the TIL program.

As the machine configuration in the Minicomputer Laboratory changes and/or new uses for the Hub are envisaged, it will be necessary to modify the Hub. The Hub was designed with two types of future modification in mind:

1) Addition of prototypes, as new applications are desired which cannot be constructed from available prototypes.
2) Moving the Hub to a different machine.

Adding Prototypes to the Hub

The procedure for adding prototypes to the Hub is best explained through an example. Consider the following application: given an input connection, write the data from that connection to some device. Call this team "file_out".

First, we must decide on the functional specification of the team. There are no output ports, as the data is output directly to the device once received. Assume that the output device is passed as an argument from the TIL program. There is one input port, for the data connection. A single process will suffice for the entire team.

It is a Hub convention that an empty buffer is appended at the end of every data stream. This can be made use of if desired. Thus, until an empty buffer is received, the file_out team must repeatedly receive buffers and output their contents to whatever device was specified. A loop to do this may be coded as follows.

```
while( buf_len = .Receive_buf(partner, buf) )
    for( i=1; i<=buf_len; ++i )  .Put( device, buf[i] );
```

Remember that .Receive_buf returns the number of data words in the buffer. The while statement will terminate when an empty buffer (zero data words) is received.

This basic functional design must be organized into a team prototype. At this point, the team root process must be coded. As the file_out team consists of only one process, the root process will contain the above loop.

The file_out team root expects three arguments when readied: an array for its port process ids, the initiator process id, and the name of the output device to be used.

The first thing the root does when readied by the initiator is check the port process ids array. The initiator process has set up this array to show which connections were specified for this team in the TIL program. Both the mandatory and the optional connections must be checked. The function Check_connections is provided for making this check.

The team root in general must create any other processes that make up the team and place the port process ids in the port process ids array at this point. In this case, the process id of the team root is the input port process id, and must be put into the port process ids array. No other processes need to be created.

The file_out team root must also allocate two resources: access to the output device and memory for a **connections** array. The connections array has the same form as the port process ids array. The function Alloc_con_array is provided for this allocation.

If none of the steps so far has failed, the address of the connections array is sent back to the initiator process. Otherwise, a zero is sent, indicating that an error occurred in the team initialization. For the file_out team, the steps the team root goes through are coded as follows.

```
check_con = Check_connections( 1, 0, 0, 0, ids );
INPUTS[ids][0] = ID[.Active];
c = Alloc_con_array();
device = .Open( file_name, "w", NO_ABORT );
buf = .Alloc_vec( HUB_BUF_SIZE );
if( !device || !check_con )  c = 0;

.Send( init_id, c );
```

The first four arguments to Check_connections are words describing the mandatory input, mandatory output, optional input and optional output ports respectively. In these words, a 1 bit signifies such a port exists for the team.

The team root then executes the statement

```
.Receive( init_id )
```

When the initiator replies, the connections array contains the port process id of the team that the file_out team is connected to. This is the process id that is used in the calls to .Receive_buf. In the general case, there may be several process ids, which must be distributed to the team's port processes before the team as a whole can begin executing.

Now the coding for the file_out team is complete. The file_out prototype follows.

```
File_out( ids, init_id, file_name )

\    prototype for teams to do output to a file
\    1 input connection, 0 output connections
\    no optional connections

  {
    extrn .Active;
    auto device, buf, c, i, partner, check_con, buf_len;

    check_con = Check_connections( 1, 0, 0, 0, ids );
    INPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    device = .Open( file_name, "w", NO_ABORT );
    buf = .Alloc_vec( HUB_BUF_SIZE );
    if( !device || !check_con )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    partner = INPUTS[c][0];
    .Select_output( device );

    while( buf_len = .Receive_buf(partner, buf) )
        for( i=1; i<=buf_len; ++i )  .Put( buf[i] );
  }
```

This and other prototypes may be found in Appendix 3. The input port of this team will always attempt to receive more data, following some finite number of output operations. Thus, assuming that the Thoth function .Put does not block indefinitely, this team satisfies the criterion mentioned in the Reliability section for all TIL teams.

Once the code has been written, the prototype must be added to those already in the Hub. All externals (function names and new "extrn" variables) must have unique names, so that they will not conflict with externals in the Hub.

To add a prototype to the Hub, the Hub must be re-compiled. There are four steps to this.

1) Add references to the source file(s) for the prototype to the source file of the Hub. For our example, we would add the line

    %port/hub/mi/fileout

    to the file port/hub/ti/.src

2) Add any newly defined manifests (for a definition of this term, see Braga 1976) to the file port/hub/mi/manifest (if machine independent) or port/hub/ti/manifest (if specific to the TI). In our case, there were no new manifests defined.

3) Add information about the new prototype to the external variables file (port/hub/ti/extern). The contents of this file are part of Appendix 2. This involves two types of externals: those used by the new prototype (none in this case) and the information that is to go into the team table. This comprises a team name ("file_out"), a team address (File_out, the name of the team root process), the memory needed (in words), and the stack needed by the team root (in words). These last two require careful analysis to determine.

4) Re-compile the Hub. This is currently done on the Honeywell time-sharing system with the system command line

eh/eh t port/hub/ti/.src

The core image thus produced must then be moved to the Hub machine.

In addition, documentation must be provided for the new prototype. This should include a brief description of the function of the prototype, along with any assumptions made in its design. The number of mandatory and optional input and output ports must be given, as well as a description of the type of data they expect to receive (if that is necessary to the function of the prototype).

Once written, the explain file is stored in two places. First, it is stored in the Honeywell file system under a name of the form

port/expls/hub/team/<name>

where <name> is some relatively mnemonic version of the prototype name (truncated to 8 characters if longer than that). Second, the file is stored in the TI 990/10 file system under a name of the form

expls/hub/team/<full name>

where now the full prototype name (up to 32 characters) may be used.

On the TI machine, the explain files are not accessed via the Hub program, but may be accessed using the command line interpreter, or shell (Young, 1977) program available. For example, an explain file for a Hub team would be accessed by a command line of the form

expl hub team <full name>

Moving the Hub to a New Machine

Before the Hub can be moved to a new machine, Thoth must be running on that machine. Only the machine dependent parts of the Hub need be changed. These are shown in Appendix 2. Basically, machine dependent quantities for the Hub are stack sizes and memory requirements, and anything related to devices not provided by Thoth. For the most part, the Hub configuration table is composed of the standard Thoth configuration table plus whatever extra devices are necessary to allow use of the communication lines to the peripheral machines. The device handling routines for the TI 990/4 and Microdata communication lines are shown in Appendix 1. The functions .Put_cim and .Get_cim used there are part of the Thoth TI 990/10 implementation.

7. Conclusions

The major conclusion of this thesis is that the Team Interconnection Language approach to data transfer in the Hub network is a feasible one. The Hub is currently being used in combination with two peripheral machines: a Microdata 1600/30 and a Texas Instruments 990/4. Programs can be loaded into these machines from the Hub, and data exchanged between them via the Hub. The most productive application so far has been the use of the Microdata for punching paper tape.

Implementing TIL teams for the Hub has not proven difficult. The work involved ranges between a few hours and a few days. Most teams take less than a day to design, test and install. It should be emphasized that TIL teams cannot be designed without a good working knowledge of Thoth. Currently, the hardest part of the design process (after the initial functional specification) is the determination of memory and stack requirements. These must be kept as low as possible, in order that the minimum amount of memory possible be required to run the team. The class of problems solvable using TIL teams is a restricted one. Applications such as merging multiple data streams cannot be implemented as TIL teams that satisfy the conditions of the termination theorem, but the main uses of the Hub do not need this kind of ability.

Care has been taken to minimize the possiblity of deadlock. A distinction must be made between **software** and **hardware** deadlocks. The Hub can only act to prevent deadlocks occurring in the software, and does so through the design of its commands and the requirements made by the termination theorem on TIL teams.

Writing TIL programs based on the available teams is a simple process. This is partially because most applications are conceptually very simple when expressed in TIL. It is usually helpful to first construct a picture of the desired data flow. A TIL graph can then be naturally drawn from this. The TIL program follows simply once a TIL graph exists.

Macros have proven very useful. When they are used, TIL programs which are the same need not be re-entered each time they are run. A class of students may be given a macro with which to load programs into one of the peripheral machines.

However, some applications are better implemented as single Thoth programs. Programs in this category include those with large memory requirements. The Hub currently uses 24K words on the TI 990/10, leaving about 7K words in which to run TIL programs. A medium size TIL program may take up 2K or more when running. Programs with complex control structures may be more easily and clearly coded as separate programs. Since a command line interpreter, or shell, is available on the Hub machine, switching from the Hub to another program is a simple process.

8. References

Braga, R., "Eh Reference Manual", Research report. CS-76-45, University of Waterloo Dept. of Computer Science, November 1976.

Cheriton, D., M. Malcolm, L. Melen, and G. Sager, "Thoth, a Portable Real-Time System", Research Report CS-77-11, University of Waterloo Dept. of Computer Science, March, 1977.

DeRemer, F. and H. Kron, "Programming-in-the-Large versus Programming-in-the-Small", SIGPLAN notices, Vol. 10, Number 6, p. 114, June 1975.

Edwards, N. P., "The Effect of Certain Modular Design Principles On Testability", SIGPLAN notices, Vol. 10, Number 6, p. 401, June 1975.

Johnson, S. C. and B. W. Kernighan, "The Programming Language B", Bell Labs Computing Science Technical Report 8, January 1973.

Malcolm, M. and G. Sager, "The Real-Time/Minicomputer Laboratory", 7th Ontario Universities Computing Conference, Proceedings, University of Waterloo, p. 36, 1976.

Tarjan, R., "Depth-first Search and Linear Graph Algorithms", SIAM Journal of Computing, Vol. 1, Number 2, p. 146, June 1972.

Young, M., unpublished manuscript, 1977.

9. Appendix 1: Machine-independent code

```
\    machine independent manifests for the Hub

BUFFER              = MESSAGE    \ spot in PD for address of
                                 \ buffer passed in .Send_buf etc.
BUFFER_LEN          = 0    \ where in the buffer the count is kept
HUB_BUF_SIZE        = 16   \ size of buffers passed
.NULL.              = 0
NO_ABORT            = 0    \ used for "safe" allocations

\    template for team descriptor used in TIL processing
\    (offsets in nodes of linked list created in get_prototypes)

.NAME.              = 0    \ this is also used in setting up tables
.PROTO.             = 1    \ index into Team_tab
.ARG.               = 2    \ argument to pass to team
.LINK.              = 3    \ to next node
.ROOT.              = 4    \ slot for process id, used in initiator
                           \ and simulations
.IDS.               = 5    \ pointer to array used by initiator
.CONNECTION_VEC.    = 6    \ ditto
.IN_DEGREE.         = 7    \ used in simulation
.OUT_DEGREE.        = 8    \ ditto
.OUTPUT.            = 9    \ offset for output connections storage
.MAX_OUTPUTS.       = 10
.MAX_INPUTS.        = 10
\    offset for input connection storage
.INPUT.             = .OUTPUT. + 2*.MAX_OUTPUTS.
.TEAM.              = 0    \ offset in .OUTPUT. area (ptr to nodes)
.CONNECTOR.         = .MAX_OUTPUTS.   \ .OUTPUT. offset for pins
.TEAM_NODE_SIZE.    = 8 + 2*.MAX_OUTPUTS. + .MAX_INPUTS.

\    following are used when referring to Team_tab
\    note:  .NAME. is also used

.START_ADDR.        = 1
.MEM_REQUIRED.      = 2
.STACK_SIZE.        = 3

\    template for nodes in initiator list

.NEXT.              = 0
.LAST.              = 1
.ID.                = 2
.TIL_INFO.          = 3
```

```
\    miscellaneous manifests

GO_AHEAD              = 1
NO_GO                 = 0
INTERACTIVE           = 0   \ force interactive entry of TIL programs
                        \ *** MUST BE ZERO ***
MASTER_CONSOLE        = 1   \ user master console
SLAVE                 = 0   \ user terminal processes created afterward

\    used in teams

END_OF_TTY            = 0
CTRLA                 = 1
CTRLQ                 = 021
CTRLS                 = 023


\    following are used to describe graph analysis results

GRAPH_IS_VALID        = 1
GRAPH_IS_NO_GOOD      = 0

\    priority levels

INITIATOR_PRIORITY    = 5
SIMULATION_PRIORITY   = 5
TERMINAL_PRIORITY     = 6
TIL_PRIORITY          = 7

\    used in addressing the ids and connection arrays

OUTPUTS               = 0
INPUTS                = 1

\    used in Valid_connection to refer to the pieces of
\    connection lines

FIRST_TEAM            = 0
SECOND_TEAM           = 2
FIRST_PORT            = 1
SECOND_PORT           = 3
```

```
Alloc_con_array()

\    for teams to allocate the connections array that
\    they pass back to the initiator process.

  {
    auto array;

    array = .Alloc_vec( 1 );
    OUTPUTS[array] = .Alloc_vec( .MAX_OUTPUTS. );
    INPUTS[array] = .Alloc_vec( .MAX_INPUTS. );
    return( array );
  }
```

```
Alloc_successful( ? )

\    recover memory in a series of vectors if one of them
\    was not successfully allocated.

  {
    auto i, n, ret;

    n = .Nargs();
    ret = GO_AHEAD;
    for( i=1; i<=n; ++i )
        if( !.Arg(i) )
          {
            ret = NO_GO;
            break;
          }

    if( ret == NO_GO )
        for( i=1; i<=n; ++i )
            if( .Arg(i) )  .Free( .Arg(i) );

    return( ret );
  }
```

```
Check_connections( mandatory_ins, mandatory_outs,
                   optional_ins, optional_outs, ids )

\   for teams to use to verify correctness of the ids
\   array passed to them.
\   ***NOTE***  this assumes that .MAX_OUTPUTS. and
\               .MAX_INPUTS. are less than or equal
\               to the number of bits in a word on
\               the Hub machine.

{
   auto i, mand_bit, opt_bit, port;

   for( i=0; i<.MAX_OUTPUTS.; ++i )
     {
       mand_bit = (mandatory_outs >> i) & 1;
       opt_bit = (optional_outs >> i) & 1;
       port = OUTPUTS[ids][i];
       if( (port == -1 && mand_bit) ||
           (port != -1 && !(mand_bit || opt_bit )) )
           return( NO_GO );
     }
   for( i=0; i<.MAX_INPUTS.; ++i )
     {
       mand_bit = (mandatory_ins >> i) & 1;
       opt_bit = (optional_ins >> i) & 1;
       port = INPUTS[ids][i];
       if( (port == -1 && mand_bit) ||
           (port != -1 && !(mand_bit || opt_bit )) )
           return( NO_GO );
     }
   return( GO_AHEAD );
}
```

```
Delete_initiator( init_id )

\    remove the Initiator init_id from the linked list
\    of initiators.    the initiator will only be found
\    in the list if the TIL program it created is still
\    running.

  {
    extrn Initiator_list;
    auto p;

    if( .ID.[Initiator_list] == init_id )
      {
        p = Initiator_list;
        if( .NEXT.[p] )    \ if not at end of list
            .LAST.[.NEXT.[p]] = &Initiator_list;
        Initiator_list = .NEXT.[p];
        .Free( p );
      }
    else
        for( p=.NEXT.[Initiator_list]; p; p = .NEXT.[p] )
            if( .ID.[p] == init_id )
              {
                .NEXT.[.LAST.[p]] = .NEXT.[p];
                if( .NEXT.[p] )
                    .LAST.[.NEXT.[p]] = .LAST.[p];
                .Free( p );
                return;
              }
  }
```

```
Del_macro()

\    delete a particular macro from the macro file
\    by effectively making the macro name a null string

  {
    auto i, name, macro_file, user;

    name = .Alloc_str( 80, NO_ABORT );
    if( !name )
      {
        .Put_str( "memory exhausted in del_macro*n" );
        return;
      }

    .Put_str( "macro name? " );
    .Flush();
    .Get_str( name, 80 );
    if( !.Null_str( name ) )
      {
        .Lower( name );
        macro_file = .Open( TIL_MACRO_FILE, "rw", NO_ABORT );
        if( !macro_file )
          {
            .Put_str( "cannot access macro file*n" );
            .Free( name );
            return;
          }
        user = .Select_input( macro_file );
        i = Find_macro( macro_file, name );
        .Select_input( user );
        if( i != -1 )
          {
            user = .Select_output( macro_file );
            .Seek( macro_file, i, ABS_BYTE );
            .Put( 0 );
            .Select_output( user );
          }
        else
            .Put_str( "macro not found*n" );

        .Close( macro_file );
      }
    .Free( name );
  }
```

```
Expand_macro( device, line )

\    tries to expand a TIL macro
\    "line" is the command line, received from main, and
\    must contain the macro name.  if the macro
\    exists, the TIL program routine is called
\    and given the position in the macro file
\    of the text for the macro.

  {
    extrn Team_tab, Initiator;
    auto i, macro_file, user;

    macro_file = .Open( TIL_MACRO_FILE, "r", NO_ABORT );
    if( !macro_file )
      {
        .Put_str( "cannot access macro file*n" );
        return;
      }
    user = .Select_input( macro_file );
    i = Find_macro( macro_file, line );
    .Select_input( user );
    .Close( macro_file );

    if( i != -1 )      \ macro found in table
        TIL_program( device, i );
    else
        .Printf( "macro %s not found*n", line );
  }




Fill_buf( buf )

\    fills the buffer "buf" from "unit", from buf[1] to
\    buf[HUB_BUF_SIZE] using .Get
\    will pad buffer with nulls

  {
    auto i;

    BUFFER_LEN[buf] = HUB_BUF_SIZE;
    for( i=1; i<=HUB_BUF_SIZE; ++i )
        buf[i] = .Get();
  }
```

```
Find_macro( macro_file, macro_name )

\   searches the macro file for "macro_name"
\   if found, returns the byte position of
\   that name.    otherwise returns -1

  {
    auto n, p, name{32};

    .Seek( macro_file, 0, ABS_BYTE );
    p = 0;
    while( n = .Get() )
      {
        p += n+1;                \   where to seek for next macro
        n = .Where( macro_file );   \ save for possible return
        .Get_str( name, 32 );
        if( .Equal( name, macro_name ) )
            return( n );
        .Seek( macro_file, p, ABS_BYTE );
      }
    return( -1 );
  }
```

```
Find_num_proto( proto_name )

\   determines and returns the index in Team_tab corresponding
\   to proto_name, if found
\   returns -1 if proto_name not found in Team_tab

  {
    extrn Team_tab, Max_teams;
    auto i;

    for( i=0; i<Max_teams; ++i )
        if( .Equal(proto_name, .NAME.[Team_tab[i]]) )
            return( i );

    return( -1 );
  }
```

```
Free_list( head )

\    free linked list info built in get_prototypes

  {
    auto p, t;

    p = head;
    while( p != .NULL. )
      {
        .Free( .NAME.[p] );
        if( .ARG.[p] )  .Free( .ARG.[p] );
        Free_up( OUTPUTS[.IDS.[p]], INPUTS[.IDS.[p]], .IDS.[p] );
        t = p;
        p = .LINK.[ p ];
        .Free( t );
      }
  }
```

```
Free_up( ? )

\    used to free a series of vectors

  {
    auto i, n;

    n = .Nargs();
    for( i=1; i<=n; ++i )
        .Free( .Arg(i) );

  }
```

```
Free_vec( vec, num )

\    free the vectors pointed to by vec[i]
\    for i=0 to i=num-1

    while( num-- )  .Free( vec[num] );
```

```
Get_arg( spot, last, pieces, arg_addr )

\    pieces ::: vector of parsed pieces of module-naming line
\    spot   ::: current piece to examine
\    last   ::: number of pieces
\    arg_addr : address of place to put argument, if found
\       this function finds arguments assuming that they
\    are surrounded by parentheses.  empty parentheses will
\    generate a null string for an argument.

    if( spot<last && .Equal( pieces[spot], "(" ) )
      {
        if( spot < last-1 && .Equal( pieces[spot+1], ")" ) )
          {
            *arg_addr = .Alloc_vec( 40, NO_ABORT );
            if( *arg_addr )  return( spot+2 );
            else  return( 0 );
          }
        if( spot < last-2 && .Equal( pieces[spot+2], ")" ) )
          {
            *arg_addr = pieces[spot+1];
            return( spot+3 );
          }
        return( 0 );    \  failure
      }
    else
      {
        *arg_addr = 0;
        return( spot );
      }


Get_command( command_line )

\    called from User_terminal to prompt for and
\    interpret user command lines.  figures out
\    which command was being issued.
\    ***NOTE***  assumes that first_word
\                will be freed by caller

  {
    auto i, c, first_word;

    first_word = .Alloc_str( 8, NO_ABORT );

    .Put( '?' );
    .Flush();
    .Get_str( command_line );
    .Lower( command_line );
    for( i=0; c=command_line{i}; ++i )
        if( c == ' ' || i > 7 )  break;
        else
            first_word{i} = c;

    return( first_word );
  }
```

```
Get_connections( file, ptr, wait_flag )

\   this function interactively gets connection lines
\   (used in defining TIL programs) from the user
\   file -- determines whether or not input is to be
\           taken from a file (for macros)
\   ptr  -- head pointer to list of module information
\           for this program
\   wait_flag -- flag to be set if "wait" exit is taken
\   connection lines are checked for correctness both syntactically
\   and contextually (see comments with valid_connection)
\   input is terminated when a null line is entered

{
  auto line, ret;

  line = .Alloc_str( 80, NO_ABORT );

  if( !line )
    {
      .Put_str( "memory exhausted in get_cons*n" );
      return( NO_GO );
    }

  ret = GO_AHEAD;   \   innocent until proven guilty, by god!
  if( file == INTERACTIVE )
      .Put_str( "*nConnections:*nsource    dest*n" );
  repeat
    {
      if( file == INTERACTIVE )
        {
          .Put( ':' );
          .Flush();
        }
      .Get_str( line, 80 );
      if( .Null_str( line ) )    break;
      .Lower( line );
      if( .Equal( line, "wait" ) )
        {
          *wait_flag = 1;
          break;
        }

      if( !Valid_connection( line, ptr ) )
        {
          ret = NO_GO;
          break;
        }
    }

  .Free( line );
  return( ret );
}
```

```
Get_ids_array()

\    build the .IDS. structure for team nodes
\    return pointer if successful, otherwise zero

  {
    auto v;

    if( !(v = .Alloc_vec( 1, NO_ABORT )) )  return( 0 );
    if( !(OUTPUTS[v] = .Alloc_vec( .MAX_OUTPUTS., NO_ABORT )) )
      {
        .Free( v );
        return( 0 );
      }
    if( !(INPUTS[v] = .Alloc_vec( .MAX_INPUTS., NO_ABORT )) )
      {
        Free_up( OUTPUTS[v], v );
        return( 0 );
      }
    return( v );
  }
```

```
Get_prototypes( file )

\   handles the module naming stage of entry
\   of TIL programs.

\   file -- determines whether input is to be taken
\           from a file (for macros) or not

\   returns a pointer to a linked list of team descriptors
\   if successful, and returns .NULL. ( 0 ) otherwise
\   input is terminated with a null line

  {
    auto top, line, pieces, p, team_num, arg, n, i, name;

    line = .Alloc_str( 80, NO_ABORT );
    pieces = .Alloc_vec( 30, NO_ABORT );

    if( !Alloc_successful( line, pieces ) )
      {
        .Put_str( "memory exhausted in get_proto*n" );
        return( .NULL. );
      }

    top = .NULL.;
    if( file == INTERACTIVE )  .Put_str( "Team naming:*n" );
    repeat
      {
        if( file == INTERACTIVE )
          {
            .Put( ':' );
            .Flush();
          }
        .Get_str( line, 80 );
        if( .Null_str(line) )  break;
        .Lower( line );
        n = .Parse_str( line, pieces, " *t", ",()" );

\   pick out the prototype and search for it in the table

        team_num = Find_num_proto( pieces[0] );
        if( team_num < 0 )
          {
            .Printf( "unknown prototype %s*n", pieces[0] );
            goto error_ret;
          }

\   now pick out team names and arguments

        i = 1;
        while( i < n )
          {
            name = pieces[i];
```

```
                .Lower( name );
                ++i;
                i = get_arg( i, n, pieces, &arg );
                if( !i )
                  {
                    .Printf( "error in processing %s*n",
                                        name );
                    goto error_ret;
                  }

                if( i < n )
                    if( .Equal( pieces[i], "," ) )  ++i;
                    else
                      {
                        .Put_str( "missing comma*n" );
                        goto error_ret;
                      }

                if( Not_already_used( name, top ) )
                  {
                    p = .Alloc_vec( .TEAM_NODE_SIZE., NO_ABORT );
                    if( !p )
                      {
                        .Put_str( "memory exhausted (getproto)*n");
                        goto error_ret;
                      }
                    .Zap_minus_ones( p, .TEAM_NODE_SIZE. );
                    .IDS.[p] = Get_ids_array();
                      if( !.IDS.[p] )
                        {
                          .Put_str( "memory exhausted (getproto)*n");
                          .Free( p );
                          goto error_ret;
                        }
                    .NAME.[p] = name;
                    .PROTO.[p] = team_num;
                    .ARG.[p] = arg;
                    .LINK.[p] = top;
                    top = p;
                  }
                else
                    .Printf( "%s already used; will be skipped*n", name );
            }
        }

    Free_up( line, pieces );
    return( top );

error_ret:
    Free_vec( pieces, n );
    Free_up( line, pieces );
    Free_list( top );
    return( .NULL. );
  }
```

```
Gnum( str, pos )

\   builds a positive base 10 integer from str, beginning
\   at str{ pos } and going until either a non-digit is
\   encountered or the end of the string is reached

\   if a non-digit is found, -1 is returned
\   otherwise, the number built is returned

  {
    auto c, r;

    r = 0;
    while( (c=str{pos++}) != '*0' )
      {
        if( (c < '0') || (c > '9') ) return( -1 );
        r = r*10 + c - '0';
      }
    return( r );
  }
```

```
Illegal_macro_name( name )

\   make sure that null strings or Hub commands cannot
\   be used as macro names
\   ***NOTE***  in this context, 1 is an error return
\               and 0 is not

    select{ name }
      {
        case ""          :
        case "q"         :
        case "delete"    :
        case "til"       :
        case "list"      :
        case "terminal"  :
        case "teams"     :
        case "macros"    :
        case "kill"      :  return( 1 );
        default          :  return( 0 );
      }
```

```
Initiator( device, ptr, creator, wait_flag )

\   device -- device with which this console communicates
\   ptr    -- head pointer to list of team information
\   creator - process id of console creating me
\   wait_flag - whether or not console is waiting to find out
\               when i finish

\   created once a graph has been approved by the Hub, either
\   because it is a predefined macro or because it was syntactically
\   correct and the graph was valid

\   here the actual creation and readying of the team roots
\   is done, and the vectors they send to initiator are filled
\   with the process ids of their connectees

  {
    extrn Team_tab, .Active, Initiator_list;
    auto p, .out, team_ptr, init_id, init_node;
    auto j, i_pin, out_id, in_id, i_team;

    init_id = ID[.Active];
    .out = .Open( device, "w" );
    .Select_output( .out );
    init_node = .Alloc_vec( 3, NO_ABORT );
    if( !init_node )
      {
        .Put_str( "memory exhausted in initiator*n" );
        .Send( creator, 0 );
        goto done;
      }

    .Printf( "Initiator %d created for this TIL program*n",
                    init_id );
    .Flush();
    .Send( creator, 0 );
    for( p=ptr; p; p=.LINK.[p] )
      {
        team_ptr = Team_tab[.PROTO.[p]];
        Set_ids_array( p );
        .ROOT.[p] = .Create( .START_ADDR.[team_ptr],
                             TIL_PRIORITY,
                             .STACK_SIZE.[team_ptr],
                             .MEM_REQUIRED.[team_ptr] );
        if ( !.ROOT.[p] )
          {
            .Printf( "create failed on %s*n",
                              .NAME.[team_ptr] );
            .Flush();
\   at this point, all of the previously created roots are
\   receive blocked on me, so when i die, they die too
```

```
            .Free( init_node );
            goto done;
          }
        .Ready( .ROOT.[p], .IDS.[p], init_id, .ARG.[p] );
        .CONNECTION_VEC.[p] = .Receive( .ROOT.[p] );
        if( !.CONNECTION_VEC.[p] )
          {
            .Printf( "failure in team %s*n",
                              .NAME.[team_ptr] );
            if( .ARG.[p] )
                .Printf( "argument was %s*n", .ARG.[p] );
            .Flush();
            .Free( init_node );
            goto done;
          }
      }

\   do connections

    for( p=ptr; p; p=.LINK.[p] )
        for ( j=0; j<.MAX_OUTPUTS.; ++j )
          {
            i_team = p[.OUTPUT.+.TEAM.+j];
            if( i_team != -1 )
              {
                i_pin = p[.OUTPUT.+.CONNECTOR.+j];
                out_id = .IDS.[p][0][j];
                in_id = .IDS.[i_team][1][i_pin];
                .CONNECTION_VEC.[p][0][j] = in_id;
                .CONNECTION_VEC.[i_team][1][i_pin] = out_id;
              }
          }

\   link myself into the initiator list

    .ID.[init_node] = init_id;
    .NEXT.[init_node] = Initiator_list;
    .LAST.[init_node] = &Initiator_list;
    if( Initiator_list )
        .LAST.[Initiator_list] = init_node;
    .TIL_INFO.[init_node] = ptr;
    Initiator_list = init_node;

\   tell the roots to start 'em up

    for( p=ptr; p; p=.LINK.[p] )
       .Send( .ROOT.[p], 0 );

\   now wait for them to die

    for( p=ptr; p; p=.LINK.[p] )
        .Send( .ROOT.[p], 0 );
```

```
\    recover memory used for information structures

     Delete_initiator( init_id );    \ remove myself from init list
done:
     Free_list( ptr );
     if( wait_flag )  .Send( creator, 0 );
   }
```

```
Kill_TIL_program()

\    used to kill the initiator process associated with
\    a TIL program.  this will cause the death of the
\    TIL program.    Initiators are identified by their
\    process ids.

  {
    extrn Initiator_list;
    auto p, init_id, init_str{8};

    .Put_str( "Initiator id? " );
    .Flush();
    .Get_str( init_str, 8 );
    init_id = Gnum( init_str, 0 );
    if( init_id == -1 )
        .Put_str( "Illegal process id*n" );
    else
      {
        for( p=Initiator_list; p; p = .NEXT.[p] )
            if( .ID.[p] == init_id )  break;
        if( p )  \ we found the culprit
          {
            .Destroy( init_id );
            Free_list( .TIL_INFO.[p] );
            Delete_initiator( init_id );
          }
        else
            .Printf( "Initiator %d not found*n", init_id );
      }
  }
```

```
List_available_macros()

\   prints a list of the available macros
\   currently, gives only names

  {
    auto macro_file, user, n, p, name{32};

    macro_file = .Open( TIL_MACRO_FILE, "r", NO_ABORT );
    if( !macro_file )
      {
        .Put_str( "cannot access macro file*n" );
        return;
      }
    .Put_str( "*navailable macros*n" );
    .Flush();
    p = 0;
    user = .Select_input( macro_file );
    while( n = .Get() )
      {
        p += n+1;
        .Get_str( name, 32 );
        if( !.Null_str( name ) )        \ avoid deleted macros
            .Printf( "%s*n", name );
        .Seek( macro_file, p, ABS_BYTE );
      }
    .Close( macro_file );
    .Select_input( user );
  }
```

```
List_available_teams()

\   prints a list of the available teams
\   currently, gives only names

  {
    extrn Max_teams, Team_tab;
    auto i;

    .Put_str( "available teams*n" );
    for( i=0; i<Max_teams; ++i )
        .Printf( "%s*n", .NAME.[Team_tab[i]] );
  }
```

```
List_TIL_programs()

\    list information about TIL programs that are
\    currently running

  {
    extrn Initiator_list;
    auto p;

    .Put_str( "Initiator list" );
    if( Initiator_list )
      {
        .Put( '*n' );
        for( p=Initiator_list; p; p = .NEXT.[p] )
            .Printf( "init_id= %d*n", .ID.[p] );
      }
    else
        .Put_str( " is empty*n" );
  }
```

```
Main()

\    main program for the Hub
\    sets up the team table
\    and master console

  {
    Set_up_table();
    User_terminal( MASTER_DEVICE, MASTER_CONSOLE );
  }
```

```
Make_terminal()

\    function that creates Hub terminals and
\    associates them with a particular device
\    ***NOTE***  does not protect against double
\                use of device as terminal

  {
    extrn User_terminal;
    auto terminal_id, device;

    device = Alloc_str( 80, NO_ABORT );
    if( !device )
      {
        .Put_str( "out of memory, no terminal created*n" );
        return;
      }

    .Put_str( "device? " );
    .Flush();
    .Get_str( device, 80 );

    terminal_id = .Create( &User_terminal, TERMINAL_PRIORITY,
                                           TERMINAL_STACK_SIZE );
    if( !terminal_id )
      {
        .Put_str( "create failed in make_terminal*n" );
        return;
      }

    .Ready( terminal_id, device, SLAVE );
  }




Not_already_used( name, ptr )

\    determines if "name" is contained in the linked
\    list of team descriptors whose head is "ptr"
\    returns :  0 if name is found
\               1 if not found or list is empty

  {
    while( ptr != .NULL. )
      {
        if( .Equal(name, .NAME.[ptr]) )  return( 0 );
        ptr = .LINK.[ ptr ];
      }

    return( 1 );
  }
```

```
Num_of_cons( ptr, offset, limit )

\   called in Valid_graph to find out how many slots
\   are used in the connections part of a team node.
\   offset is one of .OUTPUT. or .INPUT.,
\   and limit is the corresponding maximum.

  {
    auto i, cnt;

    cnt = 0;
    for( i=0; i<limit; ++i )
        if( ptr[offset+i] != -1 )  ++cnt;
    return( cnt );
  }


.Receive_buf( sender_id, buf )

\   buffer-passing counterpart of .Receive
\   does a .Copy from BUFFER[sender] to buf
\   and returns the number of data words in
\   buf
\   commits .Suicide if the sender does not exist

  {
   extrn .Active, .Pid_base, .Idmask;
   auto i, sen;

   disable;
   sen = .Pid_base[sender_id&.Idmask];
   if ( ID[sen] != sender_id )  .Suicide();

   if ( STATUS[sen] == SEND_BLOCKED
        && BLOCKED_ON[sen] == .Active )
     {
      BLOCK_BACK[BLOCK_FWD[sen]]  =  BLOCK_BACK[sen];
      BLOCK_FWD[BLOCK_BACK[sen]]  =  BLOCK_FWD[sen];

      .Copy( buf, BUFFER[sen], BUFFER_LEN[BUFFER[sen]] );

      .Add_ready( sen );
      return( BUFFER_LEN[buf] );
     }
   BUFFER[.Active] = buf;
   BLOCKED_ON[.Active] = sen;
   STATUS[.Active] = REC_BLOCKED;

   .Block();
    return( BUFFER_LEN[buf] );
  }
```

```
.Send_buf( receiver_id, buf )

\    buffer-passing counterpart of .Send
\    major difference:  does a .Copy from buf to BUFFER[receiver]

  {
   extrn .Active, .Pid_base, .Idmask;
   auto i, rec;

   disable;
   rec = .Pid_base[receiver_id&.Idmask];
   if ( ID[rec] != receiver_id )
     {
      enable;
      return;
     }
   if ( STATUS[rec] == REC_BLOCKED
        && BLOCKED_ON[rec] == .Active )
     {
      .Copy( BUFFER[rec], buf, BUFFER_LEN[buf] );

      .Add_ready( rec );
      return;
     }
   BUFFER[.Active] = buf;
   BLOCKED_ON[.Active] = rec;
   STATUS[.Active] = SEND_BLOCKED;

   i = SENDQ_TAIL[rec];
   BLOCK_FWD[.Active] = i;
   BLOCK_BACK[.Active] = BLOCK_BACK[i];
   SENDQ_TAIL[rec] = .Active;
   BLOCK_BACK[i] = .Active;

   .Block();
  }


Set_ids_array( ptr )

\    sets up the .IDS. array so that there are non-minus-one
\    entries in the same slots as the connections were made.
\    this is so that the teams can look to see if the connections
\    were made to the correct pins.

  {
    auto i;

    for( i=0; i<.MAX_OUTPUTS.; ++i )
        OUTPUTS[.IDS.[ptr]][i] = ptr[.OUTPUT.+i];
    for( i=0; i<.MAX_INPUTS.; ++i )
        INPUTS[.IDS.[ptr]][i] = ptr[.INPUT.+i];
  }
```

```
Set_up_table()

\   called by main to set up Team_tab
\   which is used in setting up TIL programs

  {
    extrn Team_tab, Max_teams, Team_names, Team_addrs;
    extrn Mem_needed, Stack_needed;
    auto i, p;

    Team_tab = .Alloc_vec( Max_teams - 1 );

    for( i=0; i<Max_teams; ++i )
      {
        p = Team_tab[i] = .Alloc_vec( 3 );
        .NAME.[p] = Team_names[i];
        .START_ADDR.[p] = Team_addrs[i];
        .MEM_REQUIRED.[p] = Mem_needed[i];
        .STACK_SIZE.[p] = Stack_needed[i];
      }
  }
```

```
Node( in_degree, out_degree, out_neighbors )

\   simulation node:  used to discover whether
\   there are cycles in a TIL graph

  {
    auto i;

    while( in_degree > 0 )
      {
        .Receive_any();
        --in_degree;
      }
    for( i=0; i<out_degree; ++i )
        .Send( out_neighbors[i], 0 );
  }
```

```
TIL_program( device, file_pos )

\    called from Hub consoles directly or indirectly to handle
\    creation of TIL programs.    This consists of the
\    module naming part ( call to get_prototype ),
\    the connection definition part ( call to
\    get_connections) and the graph validity check
\    (call to valid_graph)
\    file_pos -- determines whether we want to take input
\                from a file (for macros) or not

  {
    extrn Team_tab, Initiator, .Active;
    auto vec, n, init_id, wait, user, .in;

    wait = 0;

get_program:
    if( wait && init_id != 0 )  .Receive( init_id );
    if( file_pos != INTERACTIVE )
      {
        .in = .Open( TIL_MACRO_FILE, "r", NO_ABORT );
        if( !.in )
          {
            .Put_str( "cannot access macro file*n" );
            return;
          }
        user = .Select_input( .in );
        .Seek( .in, file_pos, ABS_BYTE );
        if( !wait )  while( .Get() ) ;  \ get past the macro name
      }
    else
      {
        .Put_str( "TIL program entry from terminal*n*n" );
        .Flush();
      }
    wait = 0;
    vec = Get_prototypes( file_pos );
    if( vec )
      {
        if( Get_connections( file_pos, vec, &wait )
            &&  Valid_graph( vec ) )
          {
\   need to prompt for missing args here
            if( file_pos != INTERACTIVE )  .Select_input( user );
            for( n=vec; n; n=.LINK.[n] )
                if( .ARG.[n] && .Null_str( .ARG.[n] ) )
                  {
                    .Printf( "arg for %s? ",
                             .NAME.[Team_tab[.PROTO.[n]]] );
                    .Flush();
                    .Get_str( .ARG.[n], 40 );
                    if( .Null_str( .ARG.[n] ) )
                      {
                        Free_list( vec );
```

```
                        goto done;      \  quit on null string
                    }
                }
            if( file_pos != INTERACTIVE )  .Select_input( .in );
            init_id = .Create( &Initiator, INITIATOR_PRIORITY,
                                            INITIATOR_STACK_SIZE );
            if( !init_id )
              {
                .Put_str( "failed to create initiator*n" );
                Free_list( vec );
                goto done;
              }
            .Ready( init_id, device, vec, ID[.Active], wait );
            .Receive( init_id );
          }
        else  Free_list( vec );
      }
done:
    if( wait )
      {
        if( file_pos != INTERACTIVE )
          {
            file_pos = .Where( .in );
            .Close( .in );
            .Select_input( user );
          }
        goto get_program;
      }
    else
        if( file_pos != INTERACTIVE )
          {
            .Close( .in );
            .Select_input( user );
          }
  }



Tty_fill_buf( buf )

\   same as fill_buf, except that buffer-filling is
\   also terminated if a *n is encountered

  {
    auto i;

    BUFFER_LEN[buf] = 0;
    for( i=1; i<=HUB_BUF_SIZE; ++i )
      {
        buf[i] = .Get();
        ++BUFFER_LEN[buf];
        if( buf[i] == '*n' ) break;
      }
  }
```

```
User_terminal( device, master )

\    Terminal (user interface process)

  {
    auto line, .in, .out, com;

    .in = .Open( device, "r", NO_ABORT );
    if( !.in )  .Suicide();
    .out = .Open( device, "w", NO_ABORT );
    if( !.out )  .Suicide();

    .Select_input( .in );
    .Select_output( .out );
    .Printf( "Hub terminal: %s*n*n", device );
    line = .Alloc_str( 80, NO_ABORT );
    if( !line )
      {
        .Put_str( "allocation failure in listener*n" );
        .Suicide();
      }

    repeat
      {
        com = Get_command( line );
        select{ com }
          {
            case "delete"   :  Del_macro();
            case "list"     :  List_TIL_programs();
            case "til"      :  TIL_program( device, INTERACTIVE );
            case "teams"    :  List_available_teams();
            case "macros"   :  List_available_macros();
            case "terminal" :  Make_terminal();
            case "kill"     :  Kill_TIL_program();
            case "q"        :  break;
            case ""         :  next;
            default         :  Expand_macro( device, line );
          }
        .Free( com );
      }

    .Free( com );
    if( !master )  .Free( device );
  }
```

```
Valid_connection( str, ptr )

\   called from Get_connections to check the validity of the
\   connection line "str" typed in by the user

\   ptr  --  head pointer to list of information nodes for teams

\   if the line is okay, the information obtained from
\   it is added to the information being built up for
\   use by the Initiator process
\   if the line is incorrect, a zero is returned, which
\   signals Get_connections to quit

\   things that are checked here:
\     -are there four items supplied? (should be two teams
\      and two pins)
\     -are the two teams distinct?
\     -are the pins integers?
\     -were the teams defined during the module naming part?
\     -are the pins within the bounds for the specified
\      teams?
\     -are the pins currently unconnected?
\   if the answer to any of the above is "no", then an error
\   has occurred.   otherwise, the connection line is processed
\   and its information added to the data structures being built

  {
    extrn Team_tab;
    auto n, i, proto_1, proto_2, pieces, ptr_1, ptr_2;
    auto port_1, port_2, ret, msg;

    msg = 0;
    pieces = .Alloc_vec( 20, NO_ABORT );
    if( !pieces )
      {
        .Put_str( "alloc failed in validcon*n" );
        return( NO_GO );
      }

    ret = NO_GO;   \   guilty until proven innocent
    n = .Parse_str( str, pieces, "- " );
    if( n != 4 )
      {
        msg = "must have 4 parts per connection*n";
        goto done;
      }

    if( .Equal( FIRST_TEAM[pieces], SECOND_TEAM[pieces] ) )
      {
        msg = "cannot connect a team to itself*n";
        goto done;
      }
```

```
port_1 = gnum( FIRST_PORT[pieces], 0 );
port_2 = gnum( SECOND_PORT[pieces], 0 );
if( port_1 < 0  || port_2 < 0  )
  {
    msg = "non-numeric in pin designator*n";
    goto done;
  }

proto_1 = -1;
proto_2 = -1;
for( i=ptr; i; i=.LINK.[i] )
  {
    if( .Equal( FIRST_TEAM[pieces], .NAME.[i] ) )
      {
        proto_1 = .PROTO.[i];
        ptr_1 = i;
      }
    if( .Equal( SECOND_TEAM[pieces], .NAME.[i] ) )
      {
        proto_2 = .PROTO.[i];
        ptr_2 = i;
      }
  }

if( (proto_1 < 0) || (proto_2 < 0) )
  {
    msg = "unrecognized name*n";
    goto done;
  }

if( port_1 >= .MAX_OUTPUTS. )
  {
    msg = "output pin number out of range*n";
    goto done;
  }
if( port_2 >= .MAX_INPUTS. )
  {
    msg = "input pin number out of range*n";
    goto done;
  }
if( ptr_1[.OUTPUT.+.TEAM.+port_1] != -1 )
  {
    msg = "first pin already connected*n";
    goto done;
  }
if( ptr_2[.INPUT.+port_2] != -1 )
  {
    msg = "second pin already connected*n";
    goto done;
  }
ptr_2[.INPUT.+port_2] = ptr_1;
ptr_1[.OUTPUT.+.TEAM.+port_1] = ptr_2;
ptr_1[.OUTPUT.+.CONNECTOR.+port_1] = port_2;
ret = GO_AHEAD;
```

```
done:
    if( msg )  .Put_str( msg );
    Free_vec( pieces, n );
    .Free( pieces );
    return( ret );
  }



Valid_graph( ptr )

\   function to check for cycles
\   in a TIL graph by simulation.   The graph is valid
\   if all members die; the test is run at higher
\   priority so control does not return unless
\   either all parts of the graph die or some are
\   blocked

\   ptr -- head pointer to list of team information nodes

  {
    extrn .Pid_base, .Idmask, Node;
    auto p, all_dead, i_sim, j, n, ret, cnt;

    if( ptr == .NULL. )  return( GRAPH_IS_NO_GOOD );

    for( p=ptr; p; p=.LINK.[p] )
      {
        .OUT_DEGREE.[p] = Num_of_cons( p, .OUTPUT., .MAX_OUTPUTS. );
        .IN_DEGREE.[p] = Num_of_cons( p, .INPUT., .MAX_INPUTS. );
        .CONNECTION_VEC.[p] = .Alloc_vec( .MAX_OUTPUTS., NO_ABORT );
        .ROOT.[p] = .Create( &Node, SIMULATION_PRIORITY,
                                    SIMULATOR_STACK_SIZE );
        if( !.CONNECTION_VEC.[p] || !.ROOT.[p] )
          {
            .Put_str( "allocation failed in simulation*n" );
            for( j=ptr; j!=p; j=.LINK.[j] )
              {
                .Destroy( .ROOT.[j] );
                .Free( .CONNECTION_VEC.[j] );
              }
            return( GRAPH_IS_NO_GOOD );
          }
      }
```

```
    for( p=ptr; p; p=.LINK.[p] )
      {
        cnt = 0;
        for( j=0; j<.MAX_OUTPUTS.; ++j )
          {
            i_sim = p[.OUTPUT.+.TEAM.+j];
            if( i_sim != -1 )
                .CONNECTION_VEC.[p][cnt++] = .ROOT.[i_sim];
          }
      }

    for( p=ptr; p; p=.LINK.[p] )
        .Ready( .ROOT.[p], .IN_DEGREE.[p], .OUT_DEGREE.[p],
                           .CONNECTION_VEC.[p] );

\   at this point, the graph should take over and execute
\   until all members are either dead or blocked, since
\   it runs at a higher priority level

    all_dead = 1;
    for( p=ptr; p; p=.LINK.[p] )
        if( ID[.Pid_base[.ROOT.[p]&.Idmask]]  ==  .ROOT.[p] )
          {
            all_dead = 0;       \  some one is still alive
            .Destroy( .ROOT.[p] );  \  recover stack
          }
    if( all_dead )  ret = GRAPH_IS_VALID;
    else
      {
        .Put_str( "cycle found in graph*n" );
        ret = GRAPH_IS_NO_GOOD;
      }

\   recover memory used for this test

done:
    for( p=ptr; p; p=.LINK.[p] )
        .Free( .CONNECTION_VEC.[p] );

    return( ret );
  }


.Zap_minus_ones( vec, last_word )

\   fill vec with -1 to vec[last_word]

  {
    auto i;

    for( i=0; i<=last_word; ++i )  vec[i] = -1;
  }
```

10. Appendix 2: Machine-specific code

```
\    Manifests for the TI 990/10 Hub

MASTER_DEVICE           = "$tty0"
INITIATOR_STACK_SIZE    = 200
TERMINAL_STACK_SIZE     = 200
SIMULATOR_STACK_SIZE    = 100

TIL_MACRO_FILE          = "hub.macros"


\    Configuration table for the TI 990/10 Hub
\    provides connections to the Microdata
\    and TI 990/4
\    besides the usual Thoth devices

.Config_table[]:

\ INIT_FUNC, DEV_INIT_FUNC, SWAPABLE_FUNC, PATHNAME,
\ IO_PROCESS or INPUT_PROCESS, OUTPUT_PROCESS
\ GET_BLK_FUNC, PUT_BLK_FUNC, SEEK_FUNC, FLUSH_FUNC,
\ OPEN_FUNC, CLOSE_FUNC, SET_OPTION_FUNC, OPTIONS,
\ PUT_BYTE_FUNC, GET_BYTE_FUNC,
\ IO_DEV_NO or INPUT_DEV_NO, OUTPUT_DEV_NO,
\ TTY_SPEED or DISC_DRIVE

  [
    .Init_tty, .Init_cim, 0, "$tty0",
    0, 0,
    .Get_blk_tty, .Put_blk_tty, .Illegal_op, .Flush_tty,
    .Open_tty, .Close_tty, .Set_tty, 0,
    .Put_cim, .Get_cim,
    CIM1_IN, CIM1_OUT,
    S_4800
  ],

  [
    .Init_tty, .Init_733, 0, "$tty1",
    0, 0,
    .Get_blk_tty, .Put_blk_tty, .Illegal_op, .Flush_tty,
    .Open_tty, .Close_tty, .Set_tty, 0,
    .Put_733, .Get_733,
    T733_IN, T733_OUT
  ],
```

```
[
  .Init_microdata, .Init_cim, 0, "$microdata",
  0, 0,
  .Illegal_op, .Illegal_op, .Illegal_op, .Illegal_op,
  .Illegal_op, .Illegal_op, .Illegal_op, 0,
  .Put_cim, .Get_cim,
  MICRO_IN, MICRO_OUT,
  S_4800
],

[
  .Init_ti_4, .Init_cim, 0, "$ti_4",
  0, 0,
  .Illegal_op, .Illegal_op, .Illegal_op, .Illegal_op,
  .Illegal_op, .Illegal_op, .Illegal_op, 0,
  .Put_cim, .Get_cim,
  TI_4_IN, TI_4_OUT,
  S_4800
],

[
  .Init_disc, .Return_1, 0, "$disc0",
  0, 0,
  .Seekf, .Seekf, .Seekf, .Flush_disc,
  .Open_disc, .Flush_disc, .Illegal_op, 0,
  0, 0,
  DISC0, 0,
  0
],

[
  .Init_filesys, .Return_1, 0, 0,
  0, 0,
  .Next_blk_file, .Next_blk_file, .Seek_file, .Flush_file,
  .Open_file, .Close_file, .Illegal_op, 0,
  0, 0,
  DISC0, DISC0,
  0
],

[
  .Init_clock, .Init_clock_dev, 0
],

[
  .Init_death, .Return_1, 0
],

0;

.CRU_base[]:  T733_BASE, T733_BASE, CIM1_BASE, CIM1_BASE,
              -1, -1, -1, -1, -1, -1, -1, MICRO_BASE, MICRO_BASE,
              TI_4_BASE, TI_4_BASE;
```

```
\    External variables for the TI Hub

.Users_root_priority : TERMINAL_PRIORITY;
Initiator_list ;       \ head of linked list of initiators
                       \ which are still alive (thus which
                       \ TIL programs are still running)

\    information for building Team_tab

Max_teams : 9;
Team_tab ;
Team_names[] : "tty_in", "tty_out", "tee", "file_in", "file_out",
               "micro_load", "micro_io", "define", "ti_4_ldr";
Team_addrs[] : Tty_in, Tty_out, Tee, File_in, File_out,
               Microload, Micro_io, Define_a_macro, Ti_ldr;
Mem_needed[] : 300, 300, 300, 400, 400, 400, 800, 800, 800;
Stack_needed[] : 150, 150, 150, 150, 150, 150, 150, 150, 150;

\    for microdata io

Micro_input ;
Micro_output ;
Switch ;

Ti_4_input ;     \    spools for the TI 990/4 i/o
Ti_4_output ;


.Init_microdata( config_tab )

\    set up the drivers and spools for i/o
\    with the Microdata

  {
    extrn Micro_input, Micro_output;
    extrn Periph_in_driver, Periph_out_driver;

    Micro_input = .Spool( 100 );
    Micro_output = .Spool( 2 );

    .Ready( .Create( &Periph_in_driver, 1, 100),
                          config_tab, Micro_input );
    .Ready( .Create( &Periph_out_driver, 1, 100),
                          config_tab, Micro_output );
  }
```

```
.Init_ti_4( config_tab )

\   set up the drivers and spools for i/o
\   with the TI 990/4

  {
    extrn Ti_4_input, Ti_4_output;
    extrn Periph_in_driver, Periph_out_driver;

    Ti_4_input = .Spool( 100 );
    Ti_4_output = .Spool( 2 );

    .Ready( .Create( &Periph_in_driver, 1, 100),
                          config_tab, Ti_4_input );
    .Ready( .Create( &Periph_out_driver, 1, 100),
                          config_tab, Ti_4_output );
  }


Periph_in_driver( config_tab, spool )

\   input driver for Hub peripheral machines

  {
    auto  dev_no, c;

    dev_no = INPUT_DEV_NO[ config_tab ];

    repeat
      {
        c = .Get_cim( dev_no );
        .Put_spool( spool, c );
      }
  }


Periph_out_driver( config_tab, spool )

\   output driver for Hub peripheral machines

  {
    auto  dev_no, c;

    dev_no = OUTPUT_DEV_NO[ config_tab ];

    repeat
      {
        c = .Get_spool( spool );
        .Put_cim( dev_no, c );
      }
  }
```

11. Appendix 3: Prototypes

```
Control( pgm_rdr, data_rdr, tty )

\   synchronizer for the Micro_io team.   Microdata uld
\   abs ldr sends 2 ctrl-s's, one ctrl-q, loads the program
\   and then sends 2 more ctrl-s's.  this explains the
\   kluge for the CTRLS case.
\   control also puts to the specified tty all characters
\   except ctrl-s and ctrl-q.

  {
    extrn Micro_input, Switch;
    auto u, c, ctrls_cnt, current_rdr;

    u = .Open( tty, "w", NO_ABORT );
    .Send( data_rdr, u );      \ tell root of team about access
    if( !u )  .Suicide();

    .Select_output( u );
    ctrls_cnt = 0;
    current_rdr = pgm_rdr;
    repeat
      {
        c = .Get_spool( Micro_input );
        select( c & $7F )
          {
            case CTRLA :  .Destroy( data_rdr );
            case CTRLQ :
              {
                Switch = 0;
                .Send( current_rdr, 0 );
              }
            case CTRLS :
              {
                Switch = 1;
                if( ctrls_cnt <= 4 )  ++ctrls_cnt;
                if( ctrls_cnt == 4 )
                  {
                    .Destroy( pgm_rdr );
                    .Send( data_rdr, 0 );
                    current_rdr = data_rdr;
                  }
              }
            default     : {
                            .Put( c );
                            .Flush();
                          }
          }
      }
  }
```

```
Define_a_macro( ids, init_id, tty )

\    team for defining new macros.  Asks for macro name
\    and then follows same pattern as TIL program writing
\    without the checks for validity.  These will be done
\    when the macro is expanded.

  {
    auto i, p1, p2, n, in, out, macro_file, line, c;
    auto check_con, header[1];

    check_con = Check_connections( 0, 0, 0, 0, ids );
    line = .Alloc_str( 80 );
    in = .Open( tty, "r", NO_ABORT );
    out = .Open( tty, "w", NO_ABORT );
    macro_file = .Open( TIL_MACRO_FILE, "rw", NO_ABORT );
    c = Alloc_con_array();
    if( !in || !out || !macro_file || !check_con )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    header[0] = "Team naming:*n";
    header[1] = "*nConnections:*nsource    dest*n";
\   find end of data in macro file
    .Select_input( macro_file );
    while( n = .Get() )
        .Seek( macro_file, n, REL_BYTE );
    p1 = .Where( macro_file ) - 1;

    .Select_output( out );
    .Select_input( in );
    .Put_str( "macro name? " );
    .Flush();
    .Get_str( line, 80 );
    .Select_input( macro_file );
    if( Illegal_macro_name( line ) ||
        Find_macro( macro_file, line ) != -1 )
      {
        .Put_str( "macro name illegal or already used*n" );
        .Suicide();
      }

    .Select_input( in );
    .Lower( line );
    .Select_output( macro_file );
    .Put_str( macro_file, line );
    .Put( macro_file, 0 );
input:
    .Select_output( out );
    for( i=0; i<2; ++i )    \ simulate the two stages of TIL
      {                     \ program entry
        .Put_str( header[i] );
```

```
        repeat
          {
            .Select_output( out );
            .Put( ':' );
            .Flush();
            .Get_str( line, 80 );
            .Lower( line );
            .Select_output( macro_file );
            .Printf( "%s*n", line );
            if( .Null_str( line ) )  break;
            if( i == 1 && .Equal( line, "wait" ) ) goto input;
          }
      }
    p2 = .Where( macro_file );
    .Select_output( macro_file );
    .Write_nulls();
    .Seek( macro_file, p1, ABS_BYTE );
    .Put( macro_file, p2-p1-1 );
    .Select_output( out );
    .Put_str( "end of macro definition*n" );
    .Flush();
}
```

```
File_in( ids, init_id, file_name )

\    team to take input from the file "file_name"
\    has  0 input connections, 1 output connection

  {
    extrn .Active;
    auto u, b, c, partner, check_con;

    check_con = Check_connections( 0, 1, 0, 0, ids );
    OUTPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    u = .Open( file_name, "r", NO_ABORT );
    if( !u || !check_con )  c = 0;
    b = .Alloc_vec( HUB_BUF_SIZE );

    .Send( init_id, c );
    .Receive( init_id );

    partner = OUTPUTS[c][0];
    .Select_input( u );
    repeat
      {
        Fill_buf( b );
        .Send_buf( partner, b );
        if ( .End_of_blks( u ) )
          {
            if( BUFFER_LEN[b] )     \ send empty buffer after eof
              {
                BUFFER_LEN[b] = 0;
                .Send_buf( partner, b );
              }
            break;
          }
      }
  }
```

```
File_out( ids, init_id, file_name )

\   prototype for teams to do output to a file
\   1 input connection, 0 output connections
\   no optional connections

  {
    extrn .Active;
    auto device, buf, c, i, partner, check_con, buf_len;

    check_con = Check_connections( 1, 0, 0, 0, ids );
    INPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    device = .Open( file_name, "w", NO_ABORT );
    buf = .Alloc_vec( HUB_BUF_SIZE );
    if( !device || !check_con )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    partner = INPUTS[c][0];
    .Select_output( device );

    while( buf_len = .Receive_buf(partner, buf) )
        for( i=1; i<=buf_len; ++i )  .Put( buf[i] );
  }


Microload( ids, init_id )

\   team to load the uld abs ldr v3-1 into the Microdata.
\   has no input or output connections

  {
    extrn .Active, Micro_output;
    auto u, c, i, check_con;

    check_con = Check_connections( 0, 0, 0, 0, ids );
    c = Alloc_con_array();
    u = .Open( "tom/mdata.absldr", "r", NO_ABORT );
    if( !u || !check_con )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    .Select_input( u );
\   there are 256 bytes in the Microdata abs ldr
    for( i=0; i<256; ++i )
        .Put_spool( Micro_output, .Get() );
  }
```

```
Micro_io( ids, init_id, tty )

\    loads a microdata program and then feeds data to it,
\    also prints data sent from the microdata on a tty
\    has 2 input connections (program and data) and
\    0 output connections

  {
    extrn .Active, Micro_output, Control, Reader;
    auto b, i, c, partner, synch, u, rdr, check_con;

    check_con = Check_connections( 3, 0, 0, 0, ids );
    rdr = .Create( &Reader, 7, 100 );
    INPUTS[ids][0] = rdr;
    INPUTS[ids][1] = ID[.Active];
    synch = .Create( &Control, 7, 100 );
    .Ready( synch, rdr, ID[.Active], tty );
    i = .Receive( synch );    \ was synch able to access tty?
    c = Alloc_con_array();
    if( !i || !check_con ) c = 0;
    b = .Alloc_vec( HUB_BUF_SIZE );

    .Send( init_id, c );
    .Receive( init_id );

    partner = INPUTS[c][1];

    .Ready( rdr, INPUTS[c][0], synch );
    .Receive( synch );
    while( .Receive_buf( partner, b ) )
        for ( i=1; i<=BUFFER_LEN[b]; ++i )
          {
            .Receive( synch );
            .Put_spool( Micro_output, b[i] );
          }
  }
```

```
Reader( partner, synch )

\   part of the Micro_io team.  this is the first input
\   connection of that team, and is supposed to be
\   taking a microdata program as input.
\   uld abs ldr sends ctrl-q when ready, so must wait for it.

  {
    extrn Micro_output, Switch;
    auto i, b;

    b = .Alloc_vec( HUB_BUF_SIZE );

    .Receive( synch );    \ microdata has sent ctrl-q
    while( .Receive_buf( partner, b ) )
        for( i=1; i<=BUFFER_LEN[b]; ++i )
          {
            while( Switch ) ;  \ 3rd and 4th ctrl-s
            .Put_spool( Micro_output, b[i] );
          }
  }


Tee( ids, init_id )

\   team that takes a single input and repeats
\   it to each of two outputs

  {
    extrn .Active;
    auto in, out1, out2, b, c, check_con;

    check_con = Check_connections( 1, 3, 0, 0, ids );
    OUTPUTS[ids][0] = ID[.Active];
    OUTPUTS[ids][1] = ID[.Active];
    INPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    b = .Alloc_vec( HUB_BUF_SIZE );
    if( !check_con )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    in = INPUTS[c][0];
    out1 = OUTPUTS[c][0];
    out2 = OUTPUTS[c][1];
    repeat
      {
        .Receive_buf( in, b );
        .Send_buf( out1, b );
        .Send_buf( out2, b );
      }
  }
```

```
Ti_control( partner, current_rdr )

\   protocol handler for loading the TI.  any messages
\   from the loader are passed back via the output port

  {
    extrn Ti_4_input;
    auto b;

    b = .Alloc_vec( HUB_BUF_SIZE );
    BUFFER_LEN[b] = 1;    \ one character at a time

    repeat
      {
        b[1] = .Get_spool( Ti_4_input );
        select( b[1] & $7F )
          {
            case CTRLQ :
                .Send( current_rdr, 0 );
            case CTRLA :
                .Destroy( current_rdr );
            default    :
                if( partner != -1 )
                    .Send_buf( partner, b );
          }
      }
  }


Ti_ldr( ids, init_id, program )

\   team to load a program into the TI 990/4
\   first loads a loader via ROM loader, then
\   loads the program from the file "program"

  {
    extrn .Active, Ti_control, Ti_4_output;
    auto u, c, n, i, check_con, ldr, synch, char;
    auto partner, there_is_an_input_connection;
    auto there_is_an_output_connection, b;

    check_con = Check_connections( 0, 0, 1, 1, ids );
    if( INPUTS[ids][0] != -1 )
        there_is_an_input_connection = 1;
    else
        there_is_an_input_connection = 0;
    if( OUTPUTS[ids][0] != -1 )
        there_is_an_output_connection = 1;
    else
        there_is_an_output_connection = 0;
    ldr = .Open( "tom/ti990.4.ldr", "r", NO_ABORT );
    u = .Open( program, "r", NO_ABORT );
```

```
    synch = .Create( &Ti_control, 7, 100 );
    OUTPUTS[ids][0] = synch;
    INPUTS[ids][0] = ID[.Active];
    b = .Alloc_vec( HUB_BUF_SIZE );
    c = Alloc_con_array();
    if( !u || !check_con || !ldr || !synch )  c = 0;

    .Send( init_id, c );
    .Receive( init_id );

    .Ready( synch, OUTPUTS[c][0], ID[.Active] );
    .Receive( synch );

    .Select_input( ldr );
    while( char=.Get() )
        select( char )
          {
            case '*l' :  next;
            case '*n' :  .Put_spool( Ti_4_output, '*r' );
            default   :  .Put_spool( Ti_4_output, char );
          }
    .Close( ldr );
    .Receive( synch );    \ loader is ready for data

    .Select_input( u );
    .Seek( u, $104, ABS_BYTE );
    n = (.Get()<<8) + .Get();    \ number of bytes in program
    .Seek( u, 0, ABS_BYTE );
    .Put_spool( Ti_4_output, n>>8 );
    .Put_spool( Ti_4_output, n&0377 );
    for( i=0; i<n; ++i )
        .Put_spool( Ti_4_output, .Get() );

    if( there_is_an_input_connection )
      {
        partner = INPUTS[ids][0];
        repeat
          {
            .Receive_buf( partner, b );
            for( i=1; i<=BUFFER_LEN[b]; ++i )
              {
                .Receive( synch );
                .Put_spool( Ti_4_output, b[i] );
              }
          }
      }

    if( there_is_an_output_connection )  .Receive( synch );
}
```

```
Tty_in( ids, init_id, tty_name )

\   team to take input from a tty
\   has 0 input connections, 1 output connection

  {
    extrn .Active;
    auto u, b, c, partner, check_con;

    check_con = Check_connections( 0, 1, 0, 0, ids );
    OUTPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    u = .Open( tty_name, "r", NO_ABORT );
    if( !u || !check_con )  c = 0;
    b = .Alloc_vec( HUB_BUF_SIZE );

    .Send( init_id, c );
    .Receive( init_id );

    partner = OUTPUTS[c][0];
    .Select_input( u );
    repeat
      {
        Tty_fill_buf( b );
        if( (b[1] & 0177) == END_OF_TTY )  break;
        .Send_buf( partner, b );
      }
    BUFFER_LEN[b] = 0;    \ send null buffer at end of data
    .Send_buf( partner, b );
  }


Tty_out( ids, init_id, tty_name )

\   team to do output to a tty
\   has 1 input connection, 0 output connections

  {
    extrn .Active;
    auto u, b, i, c, partner, check_con;

    check_con = Check_connections( 1, 0, 0, 0, ids );
    INPUTS[ids][0] = ID[.Active];
    c = Alloc_con_array();
    u = .Open( tty_name, "w", NO_ABORT );
    if( !u || !check_con )  c = 0;
    b = .Alloc_vec( HUB_BUF_SIZE );

    .Send( init_id, c );
    .Receive( init_id );

    partner = INPUTS[c][0];

    .Select_output( u );
    while( .Receive_buf( partner, b ) )
        for ( i=1; i<=BUFFER_LEN[b]; ++i )  .Put( b[i] );
  }
```

12. Appendix 4: Formal TIL Definition

TIL program	:	team_naming connections
team_naming	:	list_of_p_lines <cr>
		list_of_p_lines wait
connections	:	list_of_c_lines <cr>
list_of_p_lines	:	p_line <cr>
		p_line <cr> list_of_p_lines
list_of_c_lines	:	c_line <cr>
		c_line <cr> list_of_c_lines
p_line	:	prototype_name list_of_teams
c_line	:	source_descrip separator dest_descrip
list_of_teams	:	team
		team , list of teams
source_descrip	:	name separator connector
dest_descrip	:	name separator connector
team	:	name (argument)
		name
prototype_name	:	identifier
argument	:	identifier
name	:	identifier
connector	:	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
separator	:	blank | -
identifier	:	<{ascii characters} - {separator ( ) ,}>