SEMI-AUTOMATIC IMPLEMENTATION OF NETWORK PROTOCOLS

by

Daniel A. Ford

Technical Report 86-6

February 1986

SEMI—AUTOMATIC IMPLEMENTATION OF NETWORK PROTOCOLS

By

Daniel Alexander Ford

B.Sc.(Hons.), Simon Fraser University, 1984

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

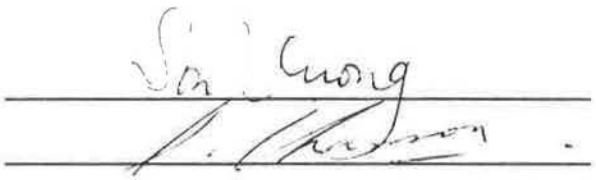
in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming

to the required standard



THE UNIVERSITY OF BRITISH COLUMBIA

March 1985

Abstract

A compiler which achieves automatic implementation of network protocols by transforming specifications written in *FDT* into *C* programs is presented. A brief introduction to the the fundamentals of *FDT*, a standard language developed by ISO/TC97/SC 16/WG 1 Subgroup B for specifying network protocols, is given. We then present an overview of the compiler and discuss the problem of *PASCAL* to *C* translation. Transformation of a *FDT* specification into code is explained and illustrated by two implementation examples. The first example illustrates the implementation strategy by tracing the processing of a simple protocol. The second example demonstrates the validity of using automatically generated implementations by showing how a communication path was established between two hosts using code generated for the alternating bit protocol.

Acknowledgement

I wish to acknowledge the assistance of my supervisor Son Vuong and the invaluable help of Jean-Marc Serre and also the financial assistance of the Natural Sciences and Engineering Research Council of Canada.

Table of Contents

Table of Figures

Chapter 1

Introduction

1.1 Motivations and Objectives

Network protocols have historically been specified in large informal documents prepared by the protocol designer. This imprecise method of specifying such a complex entity was potentially a source of problems when it came time to implement the protocol, particularly when done by someone other than its designer. The document might contain ambiguities or contradictions or it may simply be too vague. All of these inadequacies can impose implementation decisions which may or may not be compatible with decisions made for other implementations of the same protocol. Another problem of an informal specification is that it excludes the possibility of processing the protocol specification to automatically generate an implementation.

In an effort to provide more precise specification techniques, various models for protocols have been proposed [Boch 80]. The two most useful of which have been finite state machine and Petri net models. In a finite state machine model the entities in the protocol (e.g. sender, receiver) are each described as state machines. In a Petri net model the entire protocol is described in terms of a Petri net.

A practical formalism of the finite state machine model has been developed in the form of a *PASCAL* like language called *Formal Description Technique (FDT) [ISO 84]*. In addition to providing unambiguous specifications, the programming language nature of *FDT* makes it particularly amenable to processing and the automatic generation of implementations directly from their specifications.

Developing such an automatic implementation capability would remove the errors introduced by manual interpretation and implementation. Furthermore, it would allow the widespread installation of the protocol on differing machines and operating systems, subject to the availability of a translation tool (compiler) for each environment.

The motivation for this thesis is the production of such a translation tool. A tool which would correctly implement a protocol specification written in *FDT* and be as portable as possible. Achieving this goal would let installers working in different environments use the same implementation tool, thereby enhancing the compatibility of their installations and providing absolute portability of protocols at the *FDT* source code level.

A way of approaching this goal is to use a portable high level language as both the implementation and target languages of the compiler. At present, one of the most portable languages is *C* [Kern 78]. Consequently, the ultimate objective of this thesis is the production of a *FDT* compiler written in *C* which also uses *C* as its target language.

1.2 Previous Research and Thesis Contribution

There have been various attempts to produce compilers which process protocol specifications. Blummer and Tenney [Blum 81] describe one for an early version of *FDT*. Hansson [Hansson 84] describes the design of an "an integrated design environment" which provides a number of tools to aid the protocol designer and which automatically produces an implementation.

George Gerber, at the University of Montreal, has implemented a compiler in *DEC PASCAL* on a VAX-11 running VMS and which uses *DEC PASCAL* as its target language [Gerber 83]. Initial attempts by the author to port this compiler to Berkeley UNIX 4.2 BSD were unsuccessful however due to incompatibilities between Berkeley *PASCAL* and *DEC PASCAL*. While it is possible to manually alter the *FDT* compiler's source code to con-

form to Berkeley *PASCAL,* such efforts are for naught since it would still produce relatively non-portable *DEC PASCAL* code which again would require manual processing before it could be compiled.

This lack of success in porting the compiler underscores the need for a more portable tool. By using *C* as both the implementation and target language of an *FDT* compiler, it will be as portable as is presently possible.

1.3 Thesis Outline

Chapter 2 of the thesis gives an overview of the *FDT* language, Chapter 3 gives a short description of the compiler implemented and a discussion of *PASCAL* to *C* translation, Chapter 4 describes the technique used to produce an implementation from a *FDT* specification, Chapter 5 presents two implementation examples which illustrate the techniques described in Chapter 4, and finally, Chapter 6 concludes the success of the implementation and offers suggestions for further enhancements.

Chapter 2

FDT—A Formal Description Technique of Protocols

2.1 Overview

FDT (Formal Description Technique) is a language developed by Bochmann and ISO/TC97/SC16/WG1 subgroup B [ISO 84] for specifying network protocols that is based on the *PASCAL* programming language [Jensen 74]. The underlying model used in the *FDT* approach is one of communicating finite state automatons called protocol machines which exchange signals with each other and with their system environment. Except for intercommunication, the protocol machines are totally independent and operating in parallel.

FDT is really the melding of a state machine formalism and a *PASCAL* engine. This powerful combination allows the construction of extended state transition models. Such models use the variables of a programming language to augment the memory capabilities of a state machine. The net result is a powerful machine with a vastly smaller state space than would normally be possible.

In *FDT* the behaviour of a protocol machine can be specified in two ways. Either directly by a corresponding state machine described by a list of state transitions and associated actions or indirectly, by specifying a subsystem of interconnected protocol machines. The subsystem would have appropriate connections between internal interaction points and those of the enclosing machine being defined. Note that internal protocol machines are no different from any other and may themselves be specified as a composition of other state machines.

The non-procedural (i.e. non-PASCAL) part of the language is concerned with the

definition of the protocol machines and their interconnections. Four concepts are embodied in the language and are combined to produce a specification. A *module* corresponds to a type of protocol machine. A *channel* represents a communication path between two protocol machines (*modules*). A *process* is a declaration of a finite state machine. And a *refinement* is specification of protocol machine instances and their interconnection.

2.2 Module

Conceptually, each module type specified in a *FDT* specification represents one particular type of protocol machine. The actual behavior of which will be specified directly by a *process* or indirectly by a *refinement*.

As it is used in the language a *module* is a precise specification of the types (i.e. channel types) of interaction points belonging to a protocol machine. A *FDT* module specification is simply a list of interaction points. Each of which is given an identifier and a channel type in a manner analogous to a variable declaration.

The example shown in Figure 1 declares a module called *team*. This module has two interaction points called *tossin* and *tossout*, both are of channel type *toss*. The interaction point *tossin* plays the role of "catcher" and *tossout* plays the role of "thrower".

```
MODULE team;
        tossin : toss(catcher);
        tossout: toss(thrower);
end team;
```

Figure 1. Example Module Specification

2.3 Channel

In the model a channel is a communication path existing between two protocol

machines.

In a *FDT* specification, a *channel* is a precise description of the signals that may be sent via a particular interaction point by the protocol machines on either side. A *FDT* channel specification gives names to the roles played by the protocol machines (modules) on each side of the interaction point (e.g. provider/user, inputer/outputer etc.) which serve to distinguish one side from the other, and lists the signals, with any optional data fields, that can be sent by each.

In Figure 2 below is an example channel declaration for a channel type called *toss*. This channel type distinguishes the two sides of the interaction point as *thrower* and *catcher*. Where the "thrower" side is allowed to send the message *potato*, which has two integer data fields, and the "catcher" side is allowed to send the message *thanks*.

```
CHANNEL toss (thrower, catcher);

    by thrower:
        potato (thrower_team,
                thrower_num : integer);

    by catcher:
        thanks;
end toss;
```

Figure 2. Example Channel Specification

2.4 Process

A process defines an (extended) state machine and is the atomic specification of the behavior of a protocol machine.

A *FDT process* specification is a list of the transitions of a state machine, a declaration of the queueing status of each interaction point declared in the associated module, local variable declarations for the extended state model and declaration of initial values for those variables.

The queueing status of an interaction point states whether the signals sent via it are queued or not before being accepted by a protocol machine. When signals are not queued they are processed immediately upon arrival by the receiver (*rendez-vous*), even if this means suspending the processing of the sender.

Each transition listed in a *process* consists of two parts, an enabling condition and an operation. An enabling condition is a specification of:

- the present state (FROM clause)
- an input signal (WHEN clause)
- an enabling predicate (PROVIDED clause)
- a transition priority (PRIORITY clause)

Note that not all of the above need be present for every transition. In particular, if the input signal specification (WHEN clause) is absent the transition is *spontaneous*. There is also another clause, the ANY clause, which is to be used only in spontaneous transitions for selecting "random" values.

The operation portion of each transition consists of an optional specification of the next machine state (TO clause) and the action to be performed, usually a small fragment of a *PASCAL* program.

Figure 3 below contains a sample process declaration. It defines a process called *player_process* for the module type *player*. The process has two integer parameters *team_num* and *player_num* which are used to pass values to be stored as part of the state information of the state machine. The first statement declares that the two interaction points *left* and *right*, of the module *player* are to queue their signals. The next two statements declare local variables and give them initial values. The rest of the process is a list of three transition specifications. The first of which will occur whenever a *potato* signal is received via the *left*

interaction point, the second when a *thankyou* signal is received via the right interaction point, and the third occurs spontaneously.

```
PROCESS player_process(team_num, player_num : integer) FOR player;

    QUEUED left, right; (* queuing status *)

    VAR (* local variable declarations *)
     the_team, the_player : integer;

    INITIALIZE (* local variable initiallization *)
     begin
       the_team   := team_num;
       the_player := player_num;
     end;

    TRANS (* potato message on left *)
     WHEN left.potato begin
       writeln('Player ',the_player,' of team ',
            the_team,' caught a potato from player ',
            thrower_num,  ' of team ',thrower_team);
       OUT right.potato(the_team,the_player);
       OUT left.thanks;
     end;

    TRANS     (* thanks message on right *)
     WHEN right.thanks begin
       writeln('Player ', the_player,' of team ',
         the_team,' got a thankyou');
        end;

    TRANS                  (* spontaneous *)
     provided ((the_player = 1) and (the_team = 1))
      begin OUT right.potato(the_team,the_player); end;
end player_process;
```

Figure 3. Example Process Specification

2.5 Refinement

The concept of a refinement is an indirect specification of the behaviour of one protocol machine in terms of the combined behaviour of a set of other protocol machines.

A *FDT refinement* specification specifies the internal structure of a protocol machine. This includes the declaration of new, internal, channel and module types as well as processes

and other refinements. Internal refinements, in turn, can have their own internal declarations. The internal structure is defined by three sets of declarations; one for the protocol machine instances (i.e. a module and a process or a refinement), one for protocol machine interconnection (*CONNECT*) and one for the connections between internal interaction points and those of the enclosing machine being defined (*REPLACE*).

In the sample *refinement* shown in figure 4, one sees the internal declaration of a state machine's interaction points (the module *player*) and transitions (the *process player_process*). Instances of this machine, P1 and P2, are declared and then interconnected with the statment *P1.right to P2.left*. The refinement's *REPLACE* section establishes equivalence (a connection) between the interaction points *tossin* and *tossout* declared in its associated module *team* and the internal interaction points *P1.left* and *P2.right*.

```
REFINEMENT team_ref(team_num : integer) FOR team;
  MODULE player;

     left : toss(catcher);

     right: toss(thrower);
  end player;

  PROCESS player_process(team_num,
          player_num : integer) FOR player;
     (* declaration omited see figure 3 *)
  end player_process;

  (* instances *)

  P1: player with player_process(team_num, 1);
  P2: player with player_process(team_num, 2);

  CONNECT

  P1.right to P2.left;

  REPLACE

  tossin by P1.left;
  tossout by P2.right;
end team_ref;
```

Figure 4. Example Refinement Specification

Chapter 3

The FDT Compiler

3.1 The Compiler

The *FDT* compiler was developed on a VAX-11/750 running Berkeley UNIX 4.2 BSD. The implementation language used was C. Certain sections of the compiler were produced by program generators. The lexical analysis module was generated by the *LEX* [Lesk 75] utility and the parser was generated by the *YACC* [John 75] LALR parser generator utility. Both are standardly available in UNIX environments.

Not including the automatically generated code, the compiler consists of some 8000 lines of *C* and approximately 200 modules. It is structured much like all compilers, having the standard lexical analysis, parser and symbol table modules. Processing uses no intermediate forms and is completed in one pass of the source specification.

The following section discusses one of the main jobs of the compiler, the translation of *PASCAL* to *C*.

3.2 Pascal to C Translation

One of the primary issues addressed during the implementation of the compiler was the problem of translating programs and statements written in *PASCAL* to their equivalent representations in *C*. Translation between the two languages is almost straight textual substitution, they being very similar in nature having almost identical control constructs and data types. However, enough differences remain to impose limitations on the ease with which the process can be performed.

Generally, generating a *C* program from a given *PASCAL* program does not require sophisticated processing. *Begin/end* pairs are replaced by pairs of braces (i.e. { }) and data types like *reals* translate to *floats*. All of the control constructs of *PASCAL* have direct equivalents in *C*, and some even have the same names (e.g. FOR, WHILE etc.). While it is tempting to think that a text editor is more suited to the job than a powerful compiler. The language differences however preclude this possibility.

For example, the two languages have different formats for specifying I/O. *PASCAL* adopts a style similar to that of *FORTRAN* or *PL/1*, while *C* uses its own considerably different format. Direct translation between the two is difficult but not impossible, but certainly beyond the power of even a sophisticated text substitution system. A simple solution to the problem is to produce one *C* output statment (i.e. *printf*) for each field in the *PASCAL* format list.

PASCAL has a different mechanism from *C* for specifying a return value for a function. *PASCAL* uses an assignment to the function identifier and *C* uses a *return* statement. A solution to this problem is to declare a variable in the corresponding *C* function to hold the return value and then return it with a *return* statement.

The main difference between the two languages and the hardest one to overcome is their different approaches to defining the scope of identifiers. *PASCAL* views identifier scope as a variable gradation between being global to the entire program and local to a particular routine. *C* on the other hand, has only two levels of identifier scope (within one source file), external (global) and internal (local). *PASCAL* for instance allows function and procedure definitions to be nested while *C* does not. Implementing a scheme for automatically translating between the two views is not easy.

Two solutions to the problem are to use multiple output files for the generated *C* code or make all identifiers external and systematically rename them to avoid conflicts. Both will

work but, particularly in the former case, would be somewhat cumbersome to implement.

The solution adopted by this *FDT* compiler is to simply ignore the problem since the types of programs likely to be processed are unlikely to require extensive use of *PASCAL*'s scoping rules. This approach does have implications for the user as it prohibits the use of shared identifiers and global data references but should not be too much of a problem.

All in all, the problem of translating programs from *PASCAL* to *C* is not too difficult provided some restrictions on the types of programs processed can be imposed.

Chapter 4

Translating FDT Specifications into C Code

4.1 Overview

Generating a high level language program that implements a protocol specified in *FDT* is not a difficult task. The declarative part of *FDT* which specifies the protocol machines, their interconnection and the signals they pass, translates into simple conditional expressions, calls to predefined routines and structure (record) declarations. The *PASCAL* parts of the specification translate directly, requiring relatively little extra processing.

The implementation strategy adopted is the same one used successfully by Gerber in his *FDT* compiler [Gerber 83]. This approach produces two sets of routines from a *FDT* specification. One set, the *transition routines*, implements the transition processing of the protocol machines, and the other, the *initialization routines*, is responsible for constructing a data structure that represents the machines and their interconnections.

The transition routines are simply a series of conditional expressions and program blocks. The expressions evaluate the enabling conditions of state transitions and the program blocks implement the associated operations.

The initialization routines use a couple of predefined utilities to construct small parts of the data structure which are then joined to produce the complete representation.

The binding force between these two groups of routines is a couple of prewritten *driver routines* which use the initialization routines to create the necessary data structure and then the transition routines to process the transitions of the protocol machines. The driver routines are an initialization routine (*system_init*) and a transition scheduler (*schedule*) which

directs the processing of signals.

4.2 Data Structure

The data structure used to represent the protocol machines and their interconnections consists of a set of linked records. In all, four record structures are used, to store state information (*process block*), to represent an interaction point (*channel block*), to represent a signal and its data fields (*signal block*) and finally, to store index information for aggregate (array) interaction points (*index block*). When linked appropriately, records of these four types can represent arbitrarily complex *FDT* structures in a simple manner.

The representation of one protocol machine instance, and the storing of its state information, is performed by one instance of the process block record type. This record structure has fields for the major state variable (if there is one) and any local variables. It also stores the identifier of, and a pointer to, the instance's transition routine.

Process block records also contain links to other process block records and to a list of channel block records representing the protocol machine's interaction points.

Each interaction point of a protocol machine instance is represented by a channel block record. Actually a channel block record only represents half of an interaction point, the other half being represented by another channel block record belonging to the protocol machine on the other side of the interaction point. The channel blocks representing the interaction ponts of a particular machine instance are joined together to form a linked list attached to the process block of that instance.

Each channel block record stores two numbers to identify it, the first is the number of the interaction point and the second is the index of the channel block record for an aggregate interaction point. In addition, each record also contains pointers to the channel and process blocks on the other side of the communication path. These pointers constitute the actual

connection between two protocol machines.

The queue of signals which can exist at an interaction point is represented by a list of signal block records attached to a channel block record.

Signal block record instances represent messages sent by a protocol machines. They contain fields to store an identifying number and optional transmitted data. Instances are not produced by the initialization routines but come from the execution of *OUT* statements in transition routines.

Finally, those channel records belonging to aggregate interaction points store their index specified in the *FDT* source via a list of index block records. The index number stored in the channel record is the ordinal of the interaction point in a row-major ordering of the array. Index block records simply contain a single number that is the value of one dimension of an array. These values are required by the enabling conditions of transitions to identify, and possibly test, the identity of an interaction point receiving a signal.

4.3 Initialization Routines

The set of initialization routines contains two types, those that generate a representation of a machine instance (i.e. a process block and its channel list) and those that combine the machine representations into larger structures. Each process specification in the *FDT* source produces one routine belonging to the first group, and each refinement specification produces one belonging to the second. Both groups make use of a series of prewritten utilities to perform their functions.

The process initialization routines are quite simple. Their only responsibility is to construct the part of the data structure needed to represent one machine instance. They merely allocate a process block record, attach a list of channel block records, perform any initialization specified in the *FDT* source, and then add the process block record to a linked list.

The refinement initialization routines are a bit more complex as they must create a complete representation of the structures defined within them. Their task is simplified somewhat by the use of initialization routines from both groups to construct the machine instance representations. Their main job then becomes the combining of these small sections into one larger one representing the internal structure.

The operations performed by a refinement initialization routine can be divided into four phases, three of which correspond to the instance, connection and replacement sections of a *FDT* refinement and the fourth to what could be termed a "clean up" phase.

The instance section of a refinement defines the number and type of protocol machines in the internal structure. Each instance specification causes a call to the appropriate process or refinement routine to be generated, this call will simply add another machine instance on to a linked list.

The connection section of a refinement specifies which interaction points of the protocol machine instances are joined together. Each interconnection generates one call to a prewritten utility called *connect_ports* which does the actual pointer manipulation to form the connection.

The replacement section of a refinement specifies which internal interaction points are to be used for the external interaction points of the protocol machine being defined. The corresponding code in the refinement initialization routine first generates what is termed a "refinement header", consisting of a process block record and a channel block record list, and which represents the external protocol machine and its interaction points. The list of process block records produced by the instance and connection phases and which is now termed the "refinement body" is linked to the refinement header. Each replacement specification in the *FDT* source generates a call to a prewritten utility *replace_ports* which makes connections between the channel block records in the refinement header and those in the refinement

body.

The refinement header is really a place holder whose purpose is to mimic a process block and channel list produced by a process initialization routine. This uniformity in representation allows protocol machine instances from both process and refinement initialization routines to appear to be identical in the process block record list processed by the connection and replacement phases of a refinement initialization routine. This allows a consistent numbering scheme to be employed when specifying interconnections. The prewritten utilities *connect_ports* and *replace_ports* are smart enough to recognize connections to refinement headers and will make the actual connections to internal channel block records in the refinement bodies attached to the headers.

The fourth phase of a refinement routine, the "clean up" phase, is where the refinement headers holding the place of a protocol machine representation in the refinement body are removed and replaced by their refinement bodies. This task is performed by the prewritten utility *clean_up*.

4.4 Transition Routines

The actual implementation of a specified protocol, as opposed to simply creating a data structure, is achieved by a section of the generated code known as the transition routines. These routines, together with the prewritten transition scheduler, completely embody the behaviour of the model described in the *FDT* source.

The transition scheduler calls a transition routine whenever it decides that some processing should occur. The transition routine receives from the scheduler a process block record and possibly channel and signal block records. The process block record contains the state information of the machine instance needed to correctly process the state transitions. If the transition routine does receive channel and signal block records then the associated proto-

col machine has received a signal at an interaction point (the one represented by the channel block record), if the signal does not cause a transition it is simply requeued. If the routine does not receive the records then the scheduler has decided that a spontaneous transition should be attempted by the transition routine.

For their part the transition routines consist of a series of *if* statments, each of which corresponds to one transition in the *FDT* source. The boolean expression in the statement evaluates the truth value of the enabling condition and the body contains the associated action.

Chapter 5

Illustrative Implementation Examples

This chapter presents two protocols and describes their implementation. The first protocol is a simple one invented to illustrate the implementation process. The second is the alternating bit protocol.

5.1 Example 1—The Hot Potato Protocol

The simple protocol described below is an excellent vehicle for illustrating the implementation process used by the compiler. This protocol models a system of two "teams" of two "players" which continuously pass a message (a hot potato) around in a circle. Its specification contains a good sampling of the features of contained in *FDT*. In addition to the standard specification of channels, modules, processes and refinements, it contains composite protocol machine definitions and argument passing to processes and refinements.

The actions of each "player" in the protocol are identical and specified by one process. When a player receives a "potato" message from the player on his left he prints a message identifying himself and the player who sent the message. He then sends a "thankyou" message to that player and a new potato message to the player on his right.

The teams are organized identically and specified by a single refinement. The players are conceptually side-by-side, with the right of one being connected to the left of the other.

The teams are joined by connecting the two remaining sides of the players on one team to their opposite numbers on the other. Figure 5 shows the interconnection of the protocol machines involved.

As part of his local state information each player stores two numbers, one identifies his

team and the other is his player number within the team.

The complete *FDT* specification for the protocol is shown in Figures 6 and 7 below.

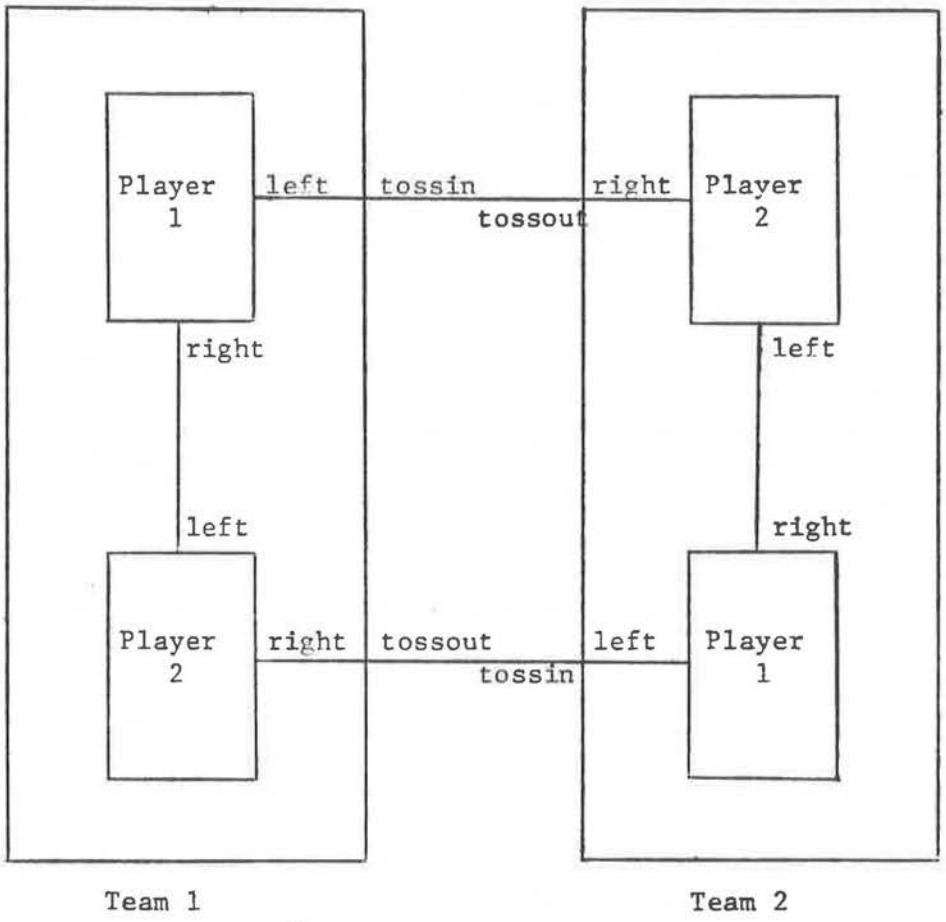


Figure 5. Illustration of Hot Potato Protocol Machines

```
MODULE hot_potato; end hot_potato;

REFINEMENT hot_ref FOR hot_potato;

    CHANNEL toss (thrower, catcher);
     by thrower: potato (thrower_team,
                thrower_num : integer);
     by catcher: thanks;
    end toss;

    MODULE team;
      tossin : toss(catcher);
      tossout: toss(thrower);
    end team;

    REFINEMENT team_ref(team_num : integer) FOR team;

      MODULE player;
          left : toss(catcher);
          right: toss(thrower);
      end player;

      PROCESS player_process(team_num,
            player_num : integer) FOR player;
          (* see Figure 7 for definition *)
      end player_process;

    P1: player with player_process(team_num, 1);
    P2: player with player_process(team_num, 2);

    CONNECT
    P1.right to P2.left;

    REPLACE
    tossin by P1.left;
    tossout by P2.right;
    end team_ref;

  T1: team with team_ref(1);
  T2: team with team_ref(2);

  CONNECT
  T1.tossout to T2.tossin;
  T2.tossout to T1.tossin;

end hot_ref;
```

Figure 6. FDT Specification of Hot Potato Protocol

```
PROCESS player_process(team_num, player_num : integer)
                        FOR player;
    QUEUED left, right; (* queuing status *)
    VAR (* local variable declarations *)
     the_team, the_player : integer;

    INITIALIZE (* local variable initiallization *)
     begin
      the_team   := team_num;
      the_player := player_num;
     end;

    TRANS (* potato message on left *)
     WHEN left.potato begin
       writeln('Player ',the_player,' of team ',
            the_team,' caught a potato from player ',
            thrower_num,  ' of team ',thrower_team);

       OUT right.potato(the_team, the_player);
       OUT left.thanks;
     end;

    TRANS    (* thanks message on right *)
     WHEN right.thanks begin
       writeln('Player ', the_player,' of team ',
         the_team,' got a thankyou');
        end;

    TRANS                (* spontaneous *)
     provided ((the_player = 1) and (the_team = 1))
      begin OUT right.potato(the_team, the_player); end;

end player_process;
```

Figure 7. FDT Specification of Player Process

The overall structure of the system is contained by the top level refinement *hot_ref* which is specified for the empty, top level, module *hot_potato*. This refinement, directly or indirectly, contains all of the other declarations. The system structure is defined by the instance and connection specifications at the end of *hot_ref*.

The refinement *team_ref* is similar to *hot_ref* except that it also has a replace specification which defines which interaction points, internal to *team_ref,* are to be used for the interaction points *tossin* and *tossout* of the refinement's module *team.* Note that *hot_ref* does not have a replace specification as its module *hot_potato* does not define any interaction points.

In the instance specifications, both refinements pass arguments to the refinement or process used. In *hot_ref* one number identifying the team of the instance is given to the refinement *team_ref,* which passes it and a player number to the process *player_process* in its instance specification. *Player_process* stores the numbers it receives as part of the local state information of each machine instance.

The actions of the players in the protocol are specified by *player_process.* It declares that the two interaction points, *left* and *right,* of its module *player* are to queue their signals, that the protocol machine has two local variables, *the_team* and *the_player,* and that they are to be initiallized to the values of the two parameters passed.

The first transition of *player_process* occurs whenever a potato message arrives via the left interaction point. It prints a message and transmits two more messages.

The second transition occurs whenever a *thanks* message arrives via the right interaction point of the protocol machine, it also prints a message.

The third transition will only occur for player one of team one when the transition scheduler detects that there are no messages enqueued and awaiting processing. For this protocol that only occurs a the start of processing.

5.1.1 Implementation Code

The implementation code produced by the compiler is placed into one output file. The

first entries in this file are a series of "#include's" whose purpose is to direct the *C* compiler to include files like the *C* standard input/output library (stdio.h). The next entries are the signal and process block declarations and then various initiallization, transition and user defined procedures which are in no particular order. The finally entry of the file is another "#include" which directs the *C* compiler to include (compile) the prewritten utility routines.

The output file generated for the example in Figures 6 and 7 is shown in Figure 8. The bodies of the procedures have been deleted for brevity and are show individually in later Figures.

```
#include <stdio.h>
#include <strings.h>
#include "listdefs.h"

struct signal_block {
      int signal_id;
      struct signal_block  *next;
      union { struct { struct {
                  int thrower_team, thrower_num;
                } potato;
            } toss;
      } lvars;
};

struct process_block {
      struct process_block *next;
      char p_ident[MAX_IDENT_LENGTH+1];
      struct channel_block *chan_list;
      struct process_block *refinement;
      int    (*proc_ptr)();
      union { struct {
                  int  the_team,  the_player;
            } s_player_process;
      } lvars;
};

struct process_block *iohot_ref(p_block)

struct process_block *ioteam_ref(p_block, team_num)

struct process_block *ioplayer_process(process_list, team_num,
                                       player_num)
player_process()

#include "fdtutil.c"
```

Figure 8. Implementation Code for Hot Potato Protocol

The signal block declaration contains two fields, *thrower_team* and *thrower_num*, which store the data values of a *potato* message. If there had been other channel declarations, at any level of nesting containing messages with data fields, they to would have appeared in the declaration.

The process_block declaration has similar fields for the storage of variables for a proto-

col machine instance. It also contains the identifier and a procedure pointer to the transition routine associated with the process_block instance.

The initiallization and transition routines follow the two structure declarations. They use the same identifiers as their associated refinement or process in the the *FDT* source, with the exception that the initiallization routines also have a "io" prefix. This prefix is necessary to distinguish between the two routines, one initiallization and one transition, generated for each process specification.

The initiallization routine generated for the refinement *hot_ref* is *iohot_ref* and is shown in Figure 9. This procedure is called by the system initiallization utility to generate the entire representation data structure.

```
struct process_block *iohot_ref(p_block)

struct process_block *p_block;
{
  struct process_block *ioteam_ref();
  struct process_block *ioteam_ref();
  struct channel_block *add_channel_block(), *c_ptr;
  struct process_block *add_refinement_header(),
                       *clean_up(), *process_list;
  process_list = NULL;
  process_list =  ioteam_ref(process_list, 1);
  process_list =  ioteam_ref(process_list, 2);

  connect_ports(process_list, 2, 2, 0, 1, 1, 0);
  connect_ports(process_list, 1, 2, 0, 2, 1, 0);

  p_block = add_refinement_header(p_block, process_list);
  p_block->refinement = clean_up(p_block->refinement);

  return(p_block);
}
```

Figure 9. Initiallization Routine for hot_ref

The initiallization routine generated for the refinement *team_ref* is shown in Figure 10. It is slightly more complex than iohot_ref as it must also add channel blocks to its refinement

header *(add_channel_block)* and establish connections between them and internal interaction points *(replace_ports)*.

The calls to *connect_ports* in Figures 9 and 10 contain a cryptic set of numbers as arguments to the routine. These numbers describe which two channel blocks are to be interconnected to form the interaction point. For example, the set of three numbers 2, 2, 0, tells *connect_ports* to look in the channel list of the second process block in the process list for a channel block containing the two numbers 2 and 0. Calls to *replace_ports* contain similar sets of numbers except that the process block number for the first channel block is not required because the channel list is know to be in the refinement header and so is omitted. Note that the numbering of the process blocks in the list is the reverse of their creation since the initiallization routines place them at the front of the process block list.

```
struct process_block *ioteam_ref(p_block, team_num)
struct process_block *p_block;
int  team_num;
{ struct process_block *ioplayer_process();
  struct process_block *ioplayer_process();
  struct channel_block *add_channel_block(), *c_ptr;
  struct process_block *add_refinement_header(),
                       *clean_up(), *process_list;

  process_list = NULL;
  process_list=ioplayer_process(process_list, team_num, 1);
  process_list=ioplayer_process(process_list, team_num, 2);

  connect_ports(process_list, 2, 2, 0, 1, 1, 0);

  p_block = add_refinement_header(p_block, process_list);
  p_block->chan_list=add_channel_block(p_block->chan_list,
                                &c_ptr, FALSE, 1, 0);
  p_block->chan_list=add_channel_block(p_block->chan_list,
                                &c_ptr, FALSE, 2, 0);

  replace_port(p_block, 1, 0, 2, 1, 0);
  replace_port(p_block, 2, 0, 1, 2, 0);

  p_block->refinement = clean_up(p_block->refinement);

  return(p_block);
}
```

Figure 10. Initialization Routine for team_ref

The two declarations of *ioplayer_process* in Figures 9 and 10 is not an error, simply a harmless idiosyncrasy of the compiler which was left in to ease the implementation of the compiler. The *C* compiler will not complain about multiple function definitions so long as they are consistent.

The initiallization routine generated for the process *player_process* is shown in Figure 11 and the transition routine in figure 12.

```
struct process_block *ioplayer_process(process_list,
                        team_num, player_num)
struct process_block *process_list;
int  team_num,  player_num;
{ struct process_block *p_block, *add_process_block();
  extern int player_process();
  struct channel_block *add_channel_block(), *c_ptr;

  p_block = add_process_block(process_list,
            player_process,"player_process");

  p_block->chan_list =
      add_channel_block(p_block->chan_list,
                  &c_ptr, TRUE, 1, 0);

  p_block->chan_list =
      add_channel_block(p_block->chan_list,
                  &c_ptr, TRUE, 2, 0);

  p_block->lvars.s_player_process.the_team =
                        team_num;

  p_block->lvars.s_player_process.the_player =
                        player_num;

  return(p_block);
}
```

Figure 11. Initiallization Routine for player_process

```
player_process(p_block, channel, signal)
  struct process_block *p_block;
  struct channel_block *channel;
  struct signal_block *signal;
{ struct signal_block *s_ptr;

  if ((channel != NULL) )
    if ((channel->c_id == 1) && (signal->signal_id == 0)) {{{
       printf("Player ");
       printf("%d",p_block->lvars.s_player_process.the_player);
       printf(" of team ");
       printf("%d",p_block->lvars.s_player_process.the_team);
       printf(" caught a potato from player ");
       printf("%d",signal->lvars.toss.potato.thrower_num);
       printf(" of team ");
       printf("%d",signal->lvars.toss.potato.thrower_team);
       printf("0);

       s_ptr = ALLOCATE(signal_block); s_ptr->signal_id = 0;
       s_ptr->lvars.toss.potato.thrower_team =
            p_block->lvars.s_player_process.the_team;
       s_ptr->lvars.toss.potato.thrower_num =
            p_block->lvars.s_player_process.the_player;
       out(p_block->chan_list, s_ptr, 2, 0);

       s_ptr = ALLOCATE(signal_block); s_ptr->signal_id = 1;
       out(p_block->chan_list, s_ptr, 1, 0);
       } goto dispose;   }}


    if ((channel != NULL) )
      if ((channel->c_id == 2) && (signal->signal_id == 1)) {{{
       printf("Player ");
       printf("%d",p_block->lvars.s_player_process.the_player);
       printf(" of team ");
       printf("%d",p_block->lvars.s_player_process.the_team);
       printf(" got a thankyou"); printf("0);
       } goto dispose;   }}

    if (channel != NULL) { requeue(channel, signal); signal = NULL;}

    if (((((p_block->lvars.s_player_process.the_player == 1)) &&
       ((p_block->lvars.s_player_process.the_team == 1))))) {{
       s_ptr = ALLOCATE(signal_block); s_ptr->signal_id = 0;
       s_ptr->lvars.toss.potato.thrower_team =
            p_block->lvars.s_player_process.the_team;
       s_ptr->lvars.toss.potato.thrower_num =
            p_block->lvars.s_player_process.the_player;
       out(p_block->chan_list, s_ptr, 2, 0);
       } goto dispose;   }
dispose: free(signal);
}
```

Figure 12. Transition Routine for player_process

5.1.2 Execution Description

The creation of the data structure that represents the interconnected protocol machines that implement the *hot_potato* protocol is initiated by a call to the initiallization routine *iohot_ref* by the prewritten utility *system_init*. The subsequent sequence of calls to the other initiallization routines and the construction of the data structure are described below.

The first operation performed by *iohot_ref* is to call the routine *ioteam_ref*, which, in turn, calls the initiallization routine *ioplayer_process* twice to create two protocol machine instances. The structure returned by one of these calls is shown in part (a) of Figure 13.

Ioteam_ref then calls *connect_ports* to connect together the interaction points of the two instances. The resulting structure is a refinement body and is shown in part (b) of Figure 13.

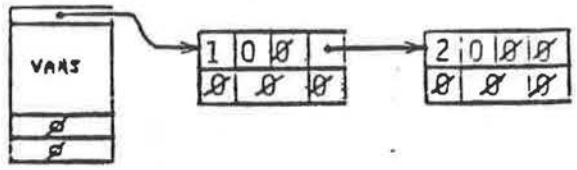
Next, *ioteam_ref* creates a refinement header *(add_refinement_header)*, attaches the refinement body and calls *replace_ports* to make interconnections between the header and the body. Its job is then completed by calling the prewritten utility *clean_up* which in this case has no work to do and then returning. The complete structure returned by *ioteam_ref* is shown in part (c) of Figure 13.

Iohot_ref now receives control and proceeds to call *ioteam_ref* a second time to generate another machine representation. That structure is shown in part (d) of Figure 13.

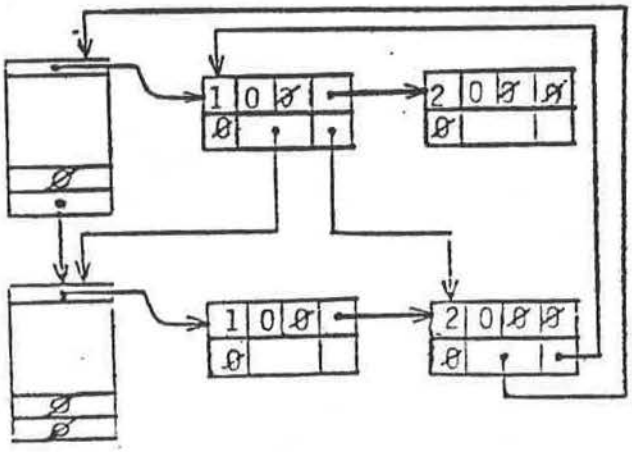
With the construction of the two instances now complete *iohot_ref* calls *connect_ports* to interconnect them. *Connect_ports* will detect that it is making connections between refinement headers and will connect to the appropriate channel blocks in their refinement bodies instead.

Iohot_ref then creates a refinement header (without channel blocks) for the refinement body (i.e. the two structures returned by *ioteam_ref*).

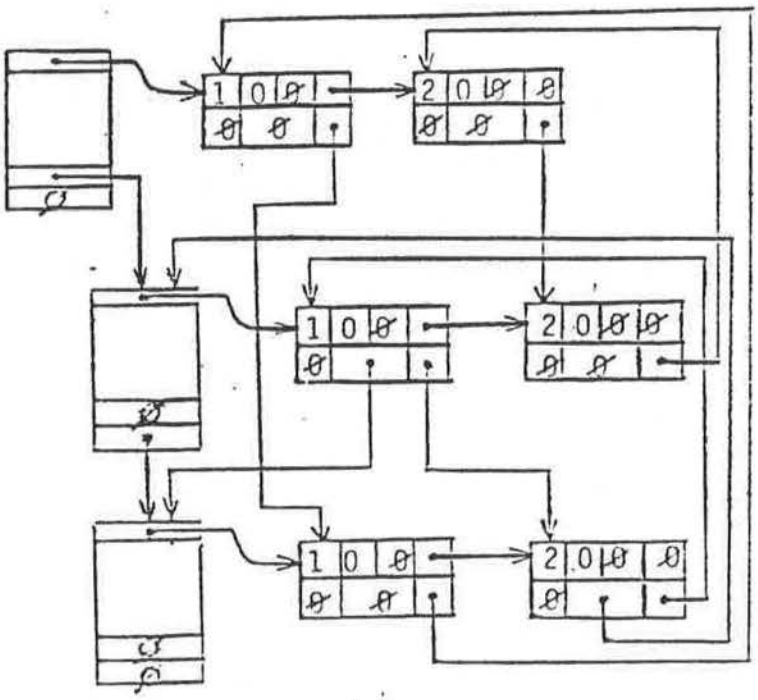
The last operation of *iohot_ref* is to call the utility *clean_up* which replaces the refinement headers, in its refinement body, with their bodies. The complete structure returned to *system_init* is shown in part (e) of Figure 13.



(a)

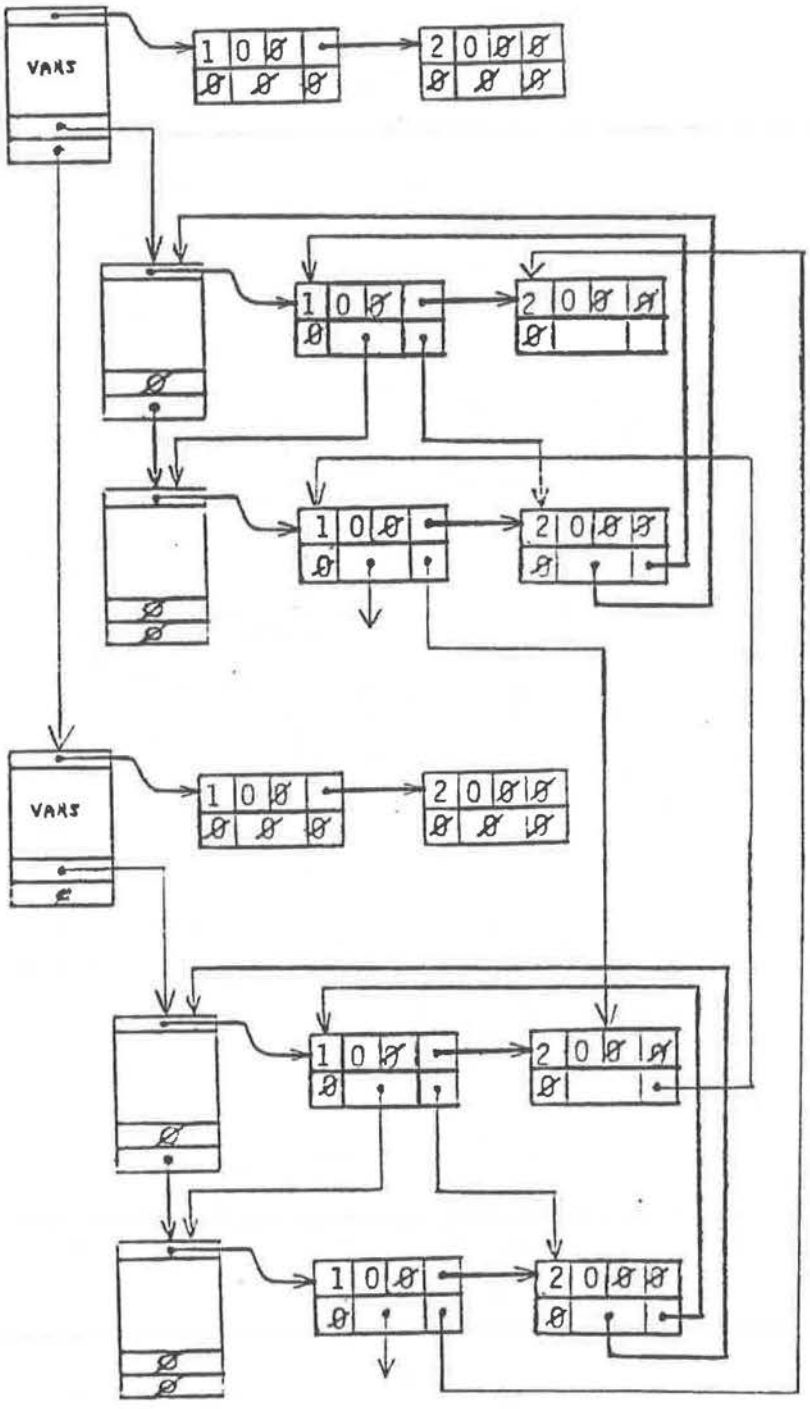


(b)



(c)

Figure 13. Representation Data Structures for Hot Potato Protocol



(d)

Figure 13. Representation Data Structures for Hot Potato Protocol (continued)

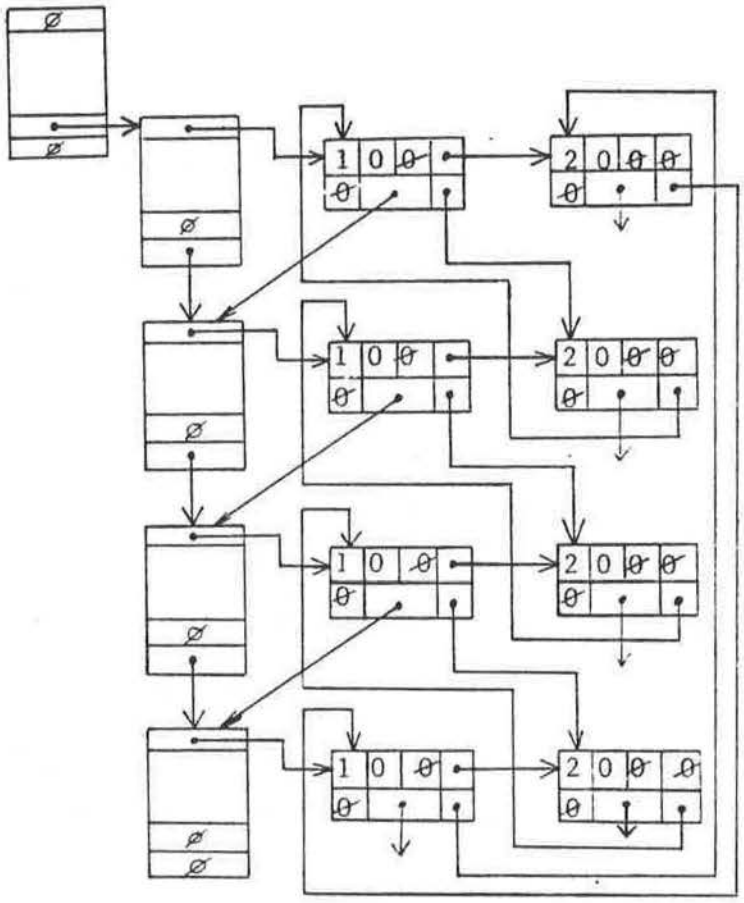


Figure 13. Representation Data Structures for Hot Potato Protocol (continued)

5.1.3 Transition Processing

When the initiallization routines have completed their task and built the representation data structure the transition scheduler receives control.

The scheduler distributes enqueued messages and spontaneous transitions in a round robin manner.

The identities of the spontaneous routines are hand coded into the scheduler by the user (not everything is automatic!). The user enters a series of string comparisons between the identifier stored in the process_block instances and the identifiers of the spontaneous transition routines. These tests are OR'ed together and become the boolean expression of an *if* statement that decides whether to call the transition routine to attempt a spontaneous transition.

In this example, when the scheduler first receives control it tests for the existence of enqueued signals. Discovering that none exist it proceeds to search the process block list for a process block that contains a transition routine identifier matching one of those known (hand coded) to contain a spontaneous transition. The first block tested just happens to contain (like the others) such an identifier *(player_process)* so the scheduler calls that routine. Once a signal does get enqueued by the spontaneous transition of the protocol machine of player one, team one, the scheduler stops searching for a spontaneous transition and goes about the job of dispatching signals.

5.2 Example 2—The Alternating Bit Protocol

The second example is an implementation of the alternating bit protocol which serves to illustrate the ability of the compiler to implement a real protocol. The specification of the protocol is taken from the ISO working document [ISO 84] and is shown in Appendix IV.

The specification contains one, top level, refinement *ab_ref*, and five module types *ab_pc*, *Alternating_Bit*, *timer*, *network* and *user*. It also has four different channel types and various messages.

The specification models the typical situation of a protocol machine on a particular level of abstraction. It has the alternating bit protocol machine and a system entity (timer) on one level, as well as entities on the levels above (user) and below (network).

This specification was used to create a communication path between two separate hosts. This required the modification of the system initialization routine *system_init*, the system scheduler *schedule* and the the two signal transmission routines *get_signal* and *out*. A *time out* facility was also implemented. Code for these routines is given in Appendix V.

The system initialization routine was modified to allow it to establish a network connection between the two hosts being used. The actual communication path was created using a set of locally developed network primitives which allow easy access to a local area network (ETHERNET).

The system scheduler was modified so that it tested the result of a call to *get_signal* before it called the destination routine.

Get_signal was modified to detect when it was at the particular channel through which signals from the network are to arrive. When at the designated channel it called a non-blocking network *receive* primitive in attempt to get a signal from the other host.

The routine *out* was modified to detect the transmission of a signal over the channel designated to carry traffic to the other host. Instead of queueing the signal it places it in a buffer and calls a network *send* primitive.

The *time out* facility was provided by a pair of *C* routines which schedule and test timers. These two routines are quite simple. The time out scheduler, *schedule_timeout*, adds the timer duration to the present time and stores the value. The routine *timer_expired* checks to see if the system time is later than the stored time and returns the appropriate

truth value.

The implementation code generated for this example can be found in Appendix VI.

Chapter 6

Conclusion

6.1 Thesis Summary

The development of formal models for protocol specification has opened the door to the possibility of automatically generating protocol implementations directly from their specification. The wide spread availability of implementation tools will allow protocol implementors to produce compatible protocol implementations with ease. The compiler developed demonstrates the feasibility of producing such a tool. It generates virtually all of the code needed to implement a protocol (the only thing it does not produce are some simple declarations and tests in two utility routines). And both it and its output code are written in highly portable *C*. Despite of its large size, 8000 lines, it runs very quickly, processing the two examples in negligible time (less than a second).

The restrictions placed on the structure of the *PASCAL* programs accepted by the compiler, namely the elimination of nested routines and global references, could be a factor when more complex protocols are to be processed. However, at present the restrictions do not appear to be too severe.

The success of the venture can be measured by contrasting automatic and manual protocol implementation techniques. The compiler produces consistently well structured and easy to understand code. The quality of manually generated code varies considerably. Compiler generated code is easy to maintain and modify, a change in the input specification is all that is required. On the other hand, manually produced code requires a great deal of effort and expense to maintain. Perhaps the greatest difference between the two implementation methods is the respective confidence given to their generated implementations. Code produced by a compiler can generally be assumed to be a correct representation of the

specification, whereas manually produced code cannot.

6.2 Enhancements

Further testing of the compiler on real-life protocols such as the ISO transport protocol is desirable. Such a test would further demonstrate the usefulness of using such tools and perhaps spawn the development of production versions of the compiler.

Enhancements to the compiler itself can also be made. The processing of the scope of transition enabling condition clauses is currently handled clumsily and as a result the usefulness of some of the shorthand clause specifications is restricted. *PASCAL* set types, were not implemented as they were not considered to be important enough to warrant the effort. The *FDT* DELAY clause which implements timer functions was not implemented as timers can be simulated with system calls. Adding the DELAY clause would enhance the compatibility of the compiler with the ISO standard.

A major enhancement to the system would be the implementation of protocol machines as separate processes running under control of the host's operating system. This approach would be a closer representation of the model of independent protocol machines then the current one.

References

[Aho 78]

Aho, A., Ullman, J., *"Principles of Compiler Design"*, Addison-Wesly Publishing Company, 1978.

[Blum 81]

Blumer, T. P., Tenney, R. L., *"An Automated Formal Specification Technique for Protocols"*, Proceedings INWG/NPL workshop, May 1981, pp. 277-326.

[Boch 80]

Bochmann, B. V., Sunshine, C. A., *"Formal Methods in Communication Protocol Design"*, *IEEE Transacations on Communications*, Vol. Com-28, No. 4, April 1980, pp. 624-631.

[Dant 80]

Danthine, A. A. S. , *"Protocol Representation with Finite-State Models"*, *IEEE Trans. Commun.* vol. COM-28, pp. 632-643, April 1980.

[Gerber 83]

Gerber, G. W. , *"Une Methode D'Implantation Automatisee de Systemes Specifies Formellement"*, Publication #142, Département D'Informatique et de Recherche Opérationnelle, Université de Montréal, Août 1983.

[Hansson 84]

Hansson, H., *"Aspie, A system for Automatic Implementation of Communication Protocols"*, *Uptec 8486R*, Uppsala Institute of Technology, Uppsala 1984.

[ISO 84]

"A Formal Description Technique based on an extended state transition model", ISO/TC 97/SC 16/WG 1 Subgroup B, Working document, March 1984.

[Jensen 74]

Jensen, K., Wirth, N., *"Pascal—User Manual and Report"*, *Lecture Notes in Computer Science* no. 18, Springer-Verlag. 1974.

[John 75]

Johnson, S. C., *"Yacc: Yet Another Compiler-Compiler"*, Comp. Sci. Tech. Rep. No. 32, Bell Labratories, Murray Hill, New Jersy 1975

[Kern 78]

Kernighan, B. W., Ritchie, D. M., *"The C Programming Language"*, Prentence-Hall, 1978.

[Lesk 75]

Lesk, M. E., *"Lex—A lexical Analyzer Generator"*, Comp. Sci. Tech. Rep. No. 39, Bell Labratories, Murray Hill, New Jersy (October 1975)

[Sunsh 79]

Sunshine, C. A. , *"Formal Techniques for Protocol Specification and Verification"*, *Computer*, vol. 12, pp. 20-27, Sept. 1979.

[Tanen 81]

TANENBAUM, A. S. , *"Computer Networks"*, Prentence-Hall, 1981.

Appendix I

FDT Grammar

```
specification  -> seqsect

seqsect      -> section ";" seqsect | /* empty */

section      -> channel | module | process | refinemt

channel      -> constd typed "channel" IDENT
                    "(" rolelist ")" ";" byclause "end" IDENT

rolelist     -> IDENT seqident

seqident     -> "," rolelist | /* empty */

byclause     -> "by" rolelist ":" signal byclause | /* empty */

signal           -> IDENT signalpara ";" signal | /* empty */

signalpara -> "(" paradef ")" | /* empty */

seqparadef -> ";" paradef | /* empty */

paradef          -> rolelist ":" basictype seqparadef

module           -> "module" IDENT ";" portlist "end" IDENT

portlist     -> rolelist ":" array IDENT "(" IDENT ")"
                    ";" portlist | /* empty */
```

```
array      -> "array" "{" indextype seqindext "}" "of"
                  | /* empty */

indextype  -> simpletype

seqindext  -> "," indextype seqindext | /* empty */

refinemt   -> "refinement" IDENT signalpara "for" IDENT ";"
                              refbody "end" IDENT

refbody    -> seqsect instance connect replace | /* empty */

instance   -> rolelist ":" IDENT "with" IDENT actualpar ";"
                              seqinst

seqinst    -> instance | /* empty */

connect    -> "connect" intconn | /* empty */

intconn    -> mport "to" mport ";" seqintconn

seqintconn -> intconn | /* empty */

replace    -> "replace" extconn | /* empty */

extconn    -> port "by" mport ";" seqextconn

seqextconn -> extconn | /* empty */

port       -> IDENT optindex

optindex   -> "{" constant listconst "}" | /* empty */

mport      -> IDENT "." port
```

```
process     -> "process" IDENT signalpara "for" IDENT ";"
                        procbody "end" IDENT

qchannel    -> "queued" rolelist ";"  |  /* empty */

procbody    -> qchannel constd typed pvard procfuncd init trans
                    |  /* empty */

pvard       -> "var" procvar  |  /* empty */

procvar     -> IDENT ":" "(" rolelist ")" ";" seqvardecl |  vardecl

stateset    -> IDENT  |  /* empty */

init        -> "initialize" stateset "begin" initstatmt
                        seqstatmt "end" ";"  |        /* empty */

initstatmt  -> plainstatmt

trans       -> "trans" seqclause opttrans

opttrans    -> trans  |  /* empty */

seqclause   -> clause seqclause  |  opttag block ";" seqtrans

clause          -> "any" paradef "do"  | "when" IDENT vparam "." IDENT
                    |  "from" rolelist  |    "to" nextmstate
                    |  "provided" expression  |  "priority" idorint

seqtrans    -> seqclause  |  /* empty */

opttag          -> IDENT ":"  |  /* empty */

vparam          -> "{" rolelist "}"  |  /* empty */
```

```
listvariable-> "," variable | /* empty */

nextmstate-> IDENT | "same"

idorint    -> IDENT | INTEGER

block      -> labeld constd typed vard procfuncd "begin"
                    statmt seqstatmt "end"

labeld     -> "label" INTEGER seqinteger ";" | /* empty */

seqinteger -> "," INTEGER seqinteger | /* empty */

constd     -> "const" defconst | /* empty */

defconst   -> IDENT "=" constant ";" seqdefconst

seqdefconst-> defconst | /* empty */

constant   -> optsign numconst | STRING

optsign    -> SIGN | /* empty */

numconst   -> INTEGER | REAL | IDENT

typed      -> "type" deftype | /* empty */

deftype    -> IDENT "=" type ";" seqdeftype

seqdeftype -> deftype | /* empty */

type       -> simpletype | optpack typstruct | "^" IDENT
            | "^" "boolean" | "^" "char"    | "^" "integer"
            | "^" "real"
```

```
simpletype -> "(" rolelist ")" | SIGN numconst ".." constant
                | INTEGER ".." constant | IDENT optconst
                | "boolean" | "char" | "integer" | "real"

optconst   -> ".." constant | /* empty */

optpack    -> "packed" | /* empty */

typstruct  -> "array" "{" simpletype seqsimplet "}" "of" type
                | "record" field "end"

seqsimplet -> "," simpletype seqsimplet | /* empty */

field      -> fixedpart seqfield
                | "case" IDENT typselect "of" variant

fixedpart  -> rolelist ":" type | /* empty */

seqfield   -> ";" field | /* empty */

typselect  -> ":" basictype | /* empty */

variant    -> constant listconst ":" "(" field ")" seqvariant
                | /* empty */

seqvariant -> ";" variant | /* empty */

listconst  -> "," constant listconst | /* empty */

vard       -> "var" vardecl | /* empty */

vardecl    -> rolelist ":" type ";" seqvardecl

seqvardecl -> vardecl | /* empty */
```

```
procfuncd -> procefuncd pfheader ";" pfbody ";" | /* empty */

pfheader  -> "procedure" IDENT lpara | "predicate" IDENT lpara
                | "function" IDENT lpara ":" basictype

pfbody        -> block | "extern" | "forward" | "primitive"

lpara     -> "(" spara seqspara ")" | /* empty */

seqspara  -> ";" spara seqspara |    /* empty */

spara     -> rolelist ":" basictype
                | "var" rolelist ":" basictype


factor    -> REAL |     STRING | "nil" | INTEGER
                | "{" seqsetint "}" | "(" expression ")"
                | "not" factor | IDENT seqfactid

seqfactid -> lseqvaria | "(" index ")"

index     -> expression seqindex

seqindex  -> "," index | /* empty */

lseqvaria -> "{" index "}" lseqvaria | "." IDENT lseqvaria
                | "^" lseqvaria | /* empty */

seqsetint -> setint lseqset | /* empty */

lseqset   -> "," setint lseqset |     /* empty */

setint    -> expression seqxpset

seqxpset  -> ".." expression | /* empty */
```

```
term        -> term OPERMULT factor | factor

simplexp    -> simplexp OPERADD term | optsign term

expression  -> simplexp OPEREL simplexp | simplexp

statmt      -> INTEGER ":" plainstatmt | plainstatmt

plainstatmt     -> IDENT appendix
                 | "out" IDENT seqindice "." IDENT actualpar
                 | "goto" INTEGER
                 | "begin" statmt seqstatmt "end"
                 | "if" expression "then" statmt else
                 | "case" expression "of" case seqcase otherw "end"
                 | "repeat" statmt seqstatmt "until" expression
                 | "while" expression "do" statmt
                 | "for" IDENT ":=" expression
                     direction expression "do" statmt
                 | "write" iolist | "writeln" lniolist
                 | /* empty */

actualpar   -> "(" index ")" | /* empty */

newmstate   -> IDENT | "same"

seqstatmt   -> ";" statmt seqstatmt | /* empty */

else        -> "else" statmt | /* empty */

seqcase     -> ";" case seqcase | /* empty */

case        -> constant listconst ":" statmt | /* empty */
```

```
otherw          -> "otherwise" statmt seqstatmt |    /* empty */

lniolist    -> iolist | /* empty */

iolist      -> "(" ioexp ")"

ioexp       -> ioident seqioident

seqioident  -> "," ioexp | /* empty */

ioident     -> expression ioextra

ioextra     -> ":" simplexp | /* empty */

direction   -> "to" | "downto"

variable    -> IDENT lseqvaria listvariable

appendix    -> actualpar | lseqvaria ":=" expression

seqindice   -> "{" index "}" seqindice | /* empty */

basictype   -> IDENT | "boolean" | "char"
                    | "integer" | "real"
```

APPENDIX II

Utility Routine Listings

```
int signal_pending;

/*-------------------------------------------------------------*/

struct      process_block    *add_process_block(process_list,proc_ptr,identifier)
/*
 This function allocates a new process block, initializes it, and places
 it at the head of the process list passed to it. The pointer to the
 process_list is returned.
*/

   struct   process_block          *process_list;
   int                             (*proc_ptr)();

   char                       *identifier;

     {
      struct      process_block          *ptr, *temp_ptr;


      ptr            = ALLOCATE( process_block);
      ptr->next      = process_list;

      ptr->proc_ptr      = proc_ptr;
      strcpy(ptr->p_ident,identifier);
      ptr->chan_list     = NULL;
      ptr->refinement    =NULL;


      return(ptr);


     }


/*-------------------------------------------------------------*/

struct      channel_block    *add_channel_block(channel_list,c_ptr,queued_flag,
```

```
                    number,index)

/*
 This function allocates a new channel block, initializes it, and places
 it at the end of the channel list passed to it. It returns the channel
 list.
*/

  struct  channel_block          *channel_list, *(*c_ptr);
  int                            queued_flag,number,index;

    {
     struct     channel_block          *ptr, *temp_ptr;


     /* make a new channel block */

     ptr            = ALLOCATE( channel_block);
     ptr->next      = NULL;
     ptr->c_id      = number;
     ptr->index_num         = index;
     ptr->signal_list  =  NULL;
     ptr->queued        = queued_flag;
     ptr->target_proc  =       NULL;
     ptr->target_channel=      NULL;
     ptr->index_list   =  NULL;

     *c_ptr = ptr;

     if (channel_list != NULL)
       {
        /* find the end of the channel list */

       for (temp_ptr = channel_list; temp_ptr->next != NULL;
               temp_ptr = temp_ptr->next);

       temp_ptr->next  =       ptr;

       return(channel_list);

       }
     else
       return(ptr);
    }


/*-------------------------------------------------------------*/


struct      channel_block    *find_channel(channel_list,id,index)
/*
 This function returns a pointer to the first channel block with the
 passed id in the channel list.
*/
```

```
   struct channel_block          *channel_list;
   int                           id, index;

     {
      struct     channel_block          *ptr;

      /* find the block */

      for (ptr = channel_list; (ptr->c_id != id)||(ptr->index_num != index);
              ptr = ptr->next);


      return(ptr);

     }


/*-------------------------------------------------------------*/


struct      process_block     *add_refinement_header(process_list,ref_body)
/*
 This function allocates a "refinement" type process header and places
 it at the head of the process list. The ref_body (also a process list)
 is then attached to the refinement header.
*/

   struct process_block      *process_list, *ref_body;

     {
      struct     process_block     *ptr,*add_process_block();


      ptr        =   add_process_block(process_list,NULL,"refinement");

      ptr->refinement=   ref_body;

      return(ptr);
     }


/*-------------------------------------------------------------*/


connect_ports(process_list, instance_num1, channel_num1,index1,
                  instance_num2, channel_num2,index2)
/*
 This routine interconnects the specified channels.
*/
   struct process_block      *process_list;
   int                       instance_num1, instance_num2,
                     channel_num1, channel_num2,
```

```
                index1, index2;

{
 struct process_block  *find_process(), *p_ptr1, *p_ptr2;
 struct channel_block  *find_channel(), *c_ptr1, *c_ptr2;

 p_ptr1          =  find_process(process_list,instance_num1);
 p_ptr2          =  find_process(process_list,instance_num2);

 c_ptr1          =  find_channel(p_ptr1->chan_list,channel_num1,
                         index1);
 c_ptr2          =  find_channel(p_ptr2->chan_list,channel_num2,
                         index2);

 /*
  If the channel blocks are already connected to another channel
  block then they must be in the header of a refinement so take
  their targets as the channel blocks to connect to.
 */

 p_ptr1        =(c_ptr1->target_proc == NULL)
               ? p_ptr1
               : c_ptr1->target_proc;

 p_ptr2        =(c_ptr2->target_proc == NULL)
               ? p_ptr2
               : c_ptr2->target_proc;

 c_ptr1        =(c_ptr1->target_channel == NULL)
               ? c_ptr1
               : c_ptr1->target_channel;

 c_ptr2        =(c_ptr2->target_channel == NULL)
               ? c_ptr2
               : c_ptr2->target_channel;


 /* make the connection */

 c_ptr1->target_proc  =  p_ptr2;
 c_ptr1->target_channel    =  c_ptr2;

 c_ptr2->target_proc  =  p_ptr1;
 c_ptr2->target_channel    =  c_ptr1;

}


/*-------------------------------------------------------------*/
```

```
struct      process_block    *find_process(process_list, number)
/*
 This function returns a pointer to the number'th element of the
 process list.
*/

   struct  process_block        *process_list;
   int                          number;


     {
      int                 i;

      for (i = 1; i < number; i++)
         process_list = process_list->next;

      return(process_list);
     }



/*-------------------------------------------------------------*/


replace_port(ref_ptr, port_num, port_index, instance, channel_num,index)
/*
 This routine connects a port (channel) in a refinement header to a
 port in the refinement body.
*/

   struct   process_block          *ref_ptr;
   int                             port_num, port_index, instance,
                           channel_num, index;
    {
     struct      process_block          *p_ptr,*find_process();
     struct      channel_block          *c_ptr1,*c_ptr2,*find_channel();


     p_ptr       =   find_process(ref_ptr->refinement,instance);

     c_ptr1          =   find_channel(ref_ptr->chan_list,
                             port_num,port_index);

     c_ptr2          =   find_channel(p_ptr->chan_list,channel_num,index);


     p_ptr       =   (c_ptr2->target_proc == NULL)
                   ? p_ptr
                   : c_ptr2->target_proc;

     c_ptr2          =   (c_ptr2->target_channel == NULL)
                   ? c_ptr2
                   : c_ptr2->target_channel;
```

```
    c_ptr1->target_proc  =  p_ptr;
    c_ptr1->target_channel    =   c_ptr2;

    c_ptr2->target_proc  =  ref_ptr;
    c_ptr2->target_channel    =   c_ptr1;

   }


/*-------------------------------------------------------------*/


struct      process_block    *clean_up(process_list)
/*
 This function removes the refinement header in a process list.  It
 returns a process list which is a concatenation of the refinement
 bodies
*/

  struct   process_block       *process_list;

    {
     struct      process_block       *ptr1, *ptr2, *remove_header();


     if (process_list == NULL)
       {
        return(NULL);
       }

     else
       {
        ptr1    =   process_list->next;
        ptr2    =   process_list;

        if (strcmp(process_list->p_ident,"refinement") == 0)
          {
           process_list      =   remove_header(process_list);

           for (ptr2 = process_list; ptr2->next != NULL;
                   ptr2 = ptr2->next);

           ptr2->next      =   ptr1;



          }

        while (ptr1 != NULL)
          {
           if (strcmp(ptr1->p_ident,"refinement") == 0)
             {
```

```
                ptr1    =   ptr1->next;
                ptr2->next  =   remove_header(ptr2->next);

                while (ptr2->next != NULL)
                  {
                   ptr2 =   ptr2->next;
                  }

                ptr2->next  =   ptr1;
               }
              else
               {
                ptr2    =   ptr1;
                ptr1    =   ptr1->next;
               }
             }
            return(process_list);
           }
         }
```

```
/*-------------------------------------------------------------*/


struct      process_block    *remove_header(ref_ptr)
/*
 This function removes the refinement header and its channel list and
 returns a pointer to the refinement body.
*/
   struct   process_block       *ref_ptr;

     {
      struct     process_block       *p_ptr;
      struct     channel_block       *c_ptr1, *c_ptr2;

      /* remove the channel list */

      for (c_ptr1 = ref_ptr->chan_list; c_ptr1 != NULL;
            c_ptr2 = c_ptr1, c_ptr1 = c_ptr1->next)
        {
         free(c_ptr2);
        }

      /* remove the process block */

      p_ptr           =   ref_ptr->refinement;
      free(ref_ptr);

      return(p_ptr);
     }
```

```
/*-----------------------------------------------------------*/


out(channel_list,signal,chan_num,index)
/*
 This routine dispatches a signal on the indicated channel.
*/

   struct  channel_block     *channel_list;
   struct  signal_block*signal;
   int                       chan_num, index;


     {
      extern     int signal_pending;

      struct     channel_block    *channel, *find_channel();


      channel = find_channel(channel_list,chan_num,index);


      if (channel->target_channel->queued)
        {
         signal_pending++;

         signal->next = channel->target_channel->signal_list;

         channel->target_channel->signal_list = signal;

        }
      else /* rendez-vous */

        {
         (*(channel->target_proc->proc_ptr))(channel->target_proc,
                channel,signal);
        }


     }



/*-----------------------------------------------------------*/


struct       process_block     *system_init()
/*
 This routine causes the generation of the data structure and then checks
 for dangling channel connections.
*/

  {
```

```
struct process_block   *ptr, *process_list, *remove_header();

struct channel_block   *c_ptr;


/* user included dcl */

struct process_block   *iohot_ref();



process_list = remove_header(iohot_ref(NULL));




for (ptr = process_list; ptr != NULL; ptr = ptr->next)
 {
  for (c_ptr = ptr->chan_list; c_ptr != NULL; c_ptr = c_ptr->next)
    {
     if (c_ptr->target_channel == NULL)
       {
        /* oops a dangling connection */

        fprintf(stderr,"0YSTEM INITIALIZATION ERROR: dangling");
        fprintf(stderr," channel in an instance of
                                ptr->p_ident);
        fprintf(stderr,"channel number %d, index %d0,c_ptr->c_id,
                                  c_ptr->index_num);
       }
    }
 }


/* join the ends of the process list into a loop */

for (ptr = process_list; ptr->next != NULL; ptr = ptr->next);

ptr->next =       process_list;


return(process_list);

}
```

```
/*-----------------------------------------------------------*/

schedule(process_list)
/*
 This is the main driving routine.
*/

   struct  process_block   *process_list;


     {
      extern     int    signal_pending;


      struct     channel_block    *c_ptr;
      struct     signal_block     *s_ptr, *get_signal();
      struct     process_block    *p_ptr, *p_ptr2;



      p_ptr2     =   process_list;

      p_ptr =   process_list;

      c_ptr  =   p_ptr->chan_list;

      signal_pending      = 0;



      while (TRUE)
       {
        if (signal_pending > 0)
          {
           s_ptr = get_signal(&c_ptr, &p_ptr);

           signal_pending--;


           /* call the transition routine */

           (*(p_ptr->proc_ptr))(p_ptr,c_ptr,s_ptr);


           /* move on to the next channel for the
              start of the next search */

           if (c_ptr->next != NULL)
              c_ptr = c_ptr->next;

           else
```

```
                {
                p_ptr = p_ptr->next;
                c_ptr = p_ptr->chan_list;
                }

            }


        if ((strcmp(p_ptr2->p_ident,"player_process") == 0))
                (*(p_ptr2->proc_ptr))(p_ptr2,NULL,NULL);

        p_ptr2 = p_ptr2->next;

      }
    }


/*-------------------------------------------------------------*/


struct      signal_block      *get_signal(c_ptr,p_ptr)
/*
 This function finds a pending signal on a signal queue.
*/


   struct  channel_block     *(*c_ptr);
   struct  process_block     *(*p_ptr);


     {
      struct      signal_block      *s_ptr, *s_ptr2;
      int               found=   FALSE;

      for (; !found; (*p_ptr) = (*p_ptr)->next,(*c_ptr)=(*p_ptr)->chan_list)
        {
         for (; ((*c_ptr) != NULL) && (!found);
              (*c_ptr)= (*c_ptr)->next)
           {
            if ((*c_ptr)->signal_list != NULL)
              {
               found = TRUE;
               if ((*c_ptr)->signal_list->next != NULL){
                  for (s_ptr = (*c_ptr)->signal_list,
                      s_ptr2 = s_ptr->next;
                     s_ptr2->next != NULL;
                   s_ptr = s_ptr2, s_ptr2 = s_ptr2->next);

                   s_ptr->next = NULL;
                   }
               else
                 {
```

```
                s_ptr2              =  (*c_ptr)->signal_list;
                (*c_ptr)->signal_list =  NULL;

              }
             goto end;

          }
        }
      }
   end:
      return(s_ptr2);

     }

/*-------------------------------------------------------------*/


add_index_block (channel, index)
/*
  This routine appends an index block on to the channels index list.
*/

   struct  channel_block   *channel;
   int                     index;


     {
      struct      index_block *ptr, *ptr2;


      ptr  =  ALLOCATE(index_block);

      ptr->next  =  NULL;
      ptr->num   =  index;

      if (channel->index_list == NULL)
         channel->index_list = ptr;

      else{
        for (ptr2 = channel->index_list; ptr2->next != NULL;
              ptr2 = ptr2->next);

        ptr2->next = ptr;
        }

     }


/*-------------------------------------------------------------*/

int get_index(channel, index_pos)
/*
  This routine retrives the index value store in an index block
```

```
  The index_pos is origin zero
*/

   struct  channel_block   *channel;
   int                     index_pos;


     {
      struct      index_block *ptr;


      for (ptr = channel->index_list; index_pos > 0;
              ptr = ptr->next,index_pos--);


      return(ptr->num);

     }


/*-------------------------------------------------------------*/


requeue(channel,signal)

/*
  This routine puts the signal passed on to the head of the signal queue.
*/


   struct  channel_block   *channel;
   struct  signal_block    *signal;


     {
      struct      signal_block      *ptr;

      extern      int signal_pending;


      signal_pending++;

      if (channel->signal_list != NULL).
        {
         for (ptr = channel->signal_list; ptr->next != NULL; ptr=ptr->next);


         ptr->next = signal;
        }
      else channel->signal_list = signal;

     }
```

```
/*-------------------------------------------------------------*/

int random_select(low,high)
/*
  This function returns a random integer between low and high, inclusive.
*/
   int                  low, high;

      {
       long random();
       float temp;

       temp =    random();
       temp = temp/0x7fffffff;
       return((low + ((high - low + 1) * temp))/1);
      }
```

Appendix III

Listdefs.h

```
#define MAX_IDENT_LENGTH    30

#define TRUE            1
#define FALSE           0

#define ALLOCATE(A)         (struct A *)malloc(sizeof(struct A));


struct      channel_block
  {
   struct  channel_block     *next;
   struct  signal_block*signal_list;

   struct  process_block     *target_proc;
   struct  channel_block     *target_channel;
   struct  index_block       *index_list;

   int                       queued;
   int                       c_id;
   int                       index_num;
  };

/*-------------------------------------------------------------*/


struct      index_block
  {
   struct  index_block       *next;
   int                       num;
  };
```

Appendix IV: Alternating Bit Specification

```
module ab_pc;
end ab_pc;

refinement ab_ref for ab_pc;

const
     retran_time      =    100; (* retranstission time *)



type
     data_type  =    integer;
     seq_type   =    0 .. 1;
     id_type         =    (A_DATA,ACK);

     timer_type =    (retransmit);

     ndata_type=
          record
            id   : id_type;
            data      : data_type;
            seq  : seq_type;
          end;
     msg_type  =
          record
            msgdata : data_type;
            msgseq  : seq_type;
          end;

     buffer_type=    record
                       flag : boolean;
                       msg: msg_type;
                     end;

     int_type   =    integer;


(*-------------------------------------------------------------*)

(* channel definitions *)
```

```
channel U_receive_point(sender, receiver);

      by sender:
            RECEIVE_request;

      by receiver:
            RECEIVE_response(UData: data_type);

end U_receive_point;


channel U_send_point(sender, receiver);

      by sender:
            SEND_request(UData : data_type);


end U_send_point;

(*-------------------------------------------------------*)


type
      time_struct       =     array[timer_type] of
                            record
                              flag : boolean;
                              secs : integer;
                              millisecs : integer;

                            end;


channel S_access_point(User, Provider);

      by User:
            Timer_request(name : timer_type;
                        time : integer);

      by Provider:
            Timer_response(Name: timer_type);
end S_access_point;


(*-------------------------------------------------------*)


channel N_access_point(User, Provider);

      by User:
            Data_request(id:id_type; data: data_type; seq:seq_type);

      by Provider:
```

```
        Data_response(id:id_type; data: data_type; seq:seq_type);

end N_access_point;

(*-------------------------------------------------------------*)

channel network_channel(sender,receiver);

    by sender,receiver:
        packet(id:id_type; data: data_type; seq:seq_type);

end network_channel;

(*-------------------------------------------------------------*)


(* module definitions *)


module Alternating_Bit;

    Us : U_send_point(receiver);
    ur : U_receive_point(receiver);
    N : N_access_point(User);
    S : S_access_point(User);

end Alternating_Bit;


process alternating_process for Alternating_Bit;

    QUEUED Us, Ur, N, S;

    var
      state:          (ACK_WAIT, ESTAB);
      send_seq:           seq_type;
      recv_seq:       seq_type;
      send_buffer:        buffer_type;
      recv_buffer:        buffer_type;
      p,q:                msg_type;

    function Ack_OK(id: id_type; seq,send_seq: seq_type) : boolean;
    begin
        Ack_OK := (id = ACK) and (seq = send_seq)

    end;
```

```
initialize
 begin
  state          := ESTAB;
  send_seq       := 0;
  recv_seq       := 0;
  send_buffer.flag   := FALSE;
  recv_buffer.flag   := FALSE;
 end;


(* transistions *)

trans

     from ESTAB          (* transition 1 *)
      to ACK_WAIT
        when Us.SEND_request
         provided not send_buffer.flag
     begin
      send_buffer.flag      :=   true;
      send_buffer.msg.msgdata  :=   UData;
      send_buffer.msg.msgseq   :=   send_seq;


      out N.DATA_request(A_DATA,Udata, send_seq);

      out S.TIMER_request(retransmit,retran_time)
     end;

trans


     from ESTAB, ACK_WAIT  (* transition 2 *)
      to SAME
        when Ur.RECEIVE_request
         provided  recv_buffer.flag
     begin

      out Ur.RECEIVE_response(recv_buffer.msg.msgdata);
      recv_buffer.flag := false;
     end;


trans
     from ACK_WAIT          (* transition 3 *)
      to ACK_WAIT
        when S.TIMER_response
         provided Name = retransmit
     begin

      out N.DATA_request(A_DATA,send_buffer.msg.msgdata,
                  send_buffer.msg.msgseq);
      out S.TIMER_request(retransmit,retran_time)
     end;
```

```
    trans
        from ESTAB          (* transition 4 *)
          to ESTAB
            when S.TIMER_response
              provided Name = retransmit
        begin
          (* do nothing: the message that cause this timer to be
            sent has been acknowledged. *)
            state := state
        end;

    trans
        from ACK_WAIT          (* transition 5 *)
          to ESTAB
            when N.DATA_response
              provided Ack_OK(id, seq, send_seq)
        begin
          send_buffer.flag := false;
          send_seq := (send_seq + 1) mod 2
        end;
    trans
        from ESTAB, ACK_WAIT  (* transistion 6 *)
          to SAME
            when N.DATA_response
              provided id = A_DATA
        begin

          out N.DATA_request(ACK, data, seq);

          if seq = recv_seq then
              begin
                recv_buffer.flag:=    true;
                recv_buffer.msg.msgdata:=      data;
                recv_buffer.msg.msgseq :=seq;

                recv_seq := (recv_seq + 1) mod 2
              end
        end;

end alternating_process;


(*-------------------------------------------------------------*)


(* timer module *)

module timer;
    S : S_access_point(provider);
```

```
end timer;


process timer_process for timer;

     QUEUED S;



     var

       index :        timer_type;
       a_timer:       time_struct;



procedure schedule_timeout(Var timer: time_struct; time_value: integer);
primitive;

predicate timer_expired(Var timer : time_struct);
primitive;



     initialize
      begin
       for index := retransmit to retransmit do
        begin
          a_timer[index].flag     :=   FALSE;
          a_timer[index].secs     :=   0;
          a_timer[index].millisecs      :=   0
        end
      end;



     (* transistions *)

     trans


     when S.Timer_request        (* transistion 1 *)
     begin

      schedule_timeout(a_timer[Name],Time);

     end;
```

```
      any index:timer_type do          (* transition 2 *)
        provided timer_expired(a_timer[index])

      begin
        a_timer[index].flag   :=   false;
        out S.Timer_response(index)
      end;

end timer_process;


(*-------------------------------------------------------*)


(* network module *)

module network;
      N          : N_access_point(Provider);
      send_path  : network_channel(sender);
      receive_path      : network_channel(receiver);
end network;



process net_proc for network;

QUEUED N, send_path, receive_path;


trans
      when N.Data_request
            begin
             OUT send_path.packet(id, data, seq)
            end;

trans
      when receive_path.packet
            begin
             OUT N.Data_response(id, data, seq)
            end;

end net_proc;


(*-------------------------------------------------------*)


(* user module *)

module the_user;
```

```
    Us : U_send_point(sender);
    Ur : U_receive_point(sender);
end the_user;



process user_process(number : integer) for the_user;

QUEUED Us, Ur;

type
    request_type      =    (send, receive);
    message_value_type =       1..9;



    var
      user_number : integer;


initialize
    begin
    user_number := number
    end;


trans
    when Ur.RECEIVE_response
        begin
        writeln('User ',user_number,' received the message ',
                    UData);
        end;

trans
    any request : request_type; message : message_value_type do
        provided request = send
            begin
              writeln('User ',user_number,' sent the message ',
                        message);
              OUT Us.SEND_request(message);
            end;

        provided request = receive
            begin
              OUT Ur.RECEIVE_request
            end;


end user_process;

(*-------------------------------------------------------*)

(* define some instances *)
```

```
A1 : Alternating_Bit with alternating_process;
A2 : Alternating_Bit with alternating_process;

timer1 : timer with timer_process;
timer2 : timer with timer_process;

N1 : network with net_proc;
N2 : network with net_proc;

U1 : the_user with user_process(1);
U2 : the_user with user_process(2);


(* and connect them together *)

CONNECT

A1.Us to U1.Us;
A1.Ur to U1.Ur;
A1.N to N1.N;
A1.S to timer1.S;

A2.Us to U2.Us;
A2.Ur to U2.Ur;
A2.N to N2.N;
A2.S to timer2.S;

N1.send_path to N2.receive_path;
N2.send_path to N1.receive_path;


(* no replaces *)


end ab_ref;
```

Appendix V

Utility Routines for Alternating Bit

```
#include "inet.h"

#define ROLE 1 /* or 0 */

out(channel_list,signal,chan_num,index)
/*
  This routine dispatches a signal on the indicated channel.
*/

   struct  channel_block     *channel_list;
   struct  signal_block *signal;
   int                       chan_num, index;


      {
       extern      int signal_pending, in_count, out_count;

       extern NET_CONN     conn;

       struct      channel_block    *channel, *find_channel();
       int                 n;


       channel = find_channel(channel_list,chan_num,index);

       if ((channel->target_channel != NULL) &&
            (channel->target_channel->queued) )
         {
          signal_pending++;

          signal->next = channel->target_channel->signal_list;

          channel->target_channel->signal_list = signal;

         }
       else /* send it out on the network */

         {
          buf[0] = signal->signal_id;
          buf[1] = signal->lvars.network_channel.packet.id;
          buf[2] = signal->lvars.network_channel.packet.data;
```

```
        buf[3] = signal->lvars.network_channel.packet.seq;

        free(signal);
        out_count++;


        if ((n =N_send(&conn,buf,16)) != NET_OK)
          {
           printf(">>> out : send problem %d ",n);
           exit(1);
          }

       }
    }



/*-------------------------------------------------------------*/


struct      process_block    *system_init()
/*
 This routine causes the generation of the data structure and then checks
 for dangling channel connections.
*/

  {
   struct process_block   *ptr, *process_list, *remove_header();

   struct channel_block   *c_ptr;

   int    n;


   /* user included dcl */

   struct process_block   *ioab_ref();


   if (ROLE == 1) /*server*/
    {
     n = N_open(&conn,"T_port1@cs",CLSTWO);
     if (n != NET_OK)
       {
        printf(">>> service : N_open error %d0,n);
        exit(1);
       }

     n = N_accept(&conn);
     if (n != NET_OK)
       {
        printf(">>> service : N_accept error %d0,n);
```

```
        exit(1);
      }
    }
  else
    {
    if ((n = N_connect(&conn, "T_port1@cs",CLSTWO)) != NET_OK)

      {
        printf(">>> receiver : N_connect error %d",n);
        exit(1);
      }
    }


  process_list = remove_header(ioab_ref(NULL));


  for (ptr = process_list; ptr != NULL; ptr = ptr->next)
    {
     for (c_ptr = ptr->chan_list; c_ptr != NULL; c_ptr = c_ptr->next)
       {
        if (c_ptr->target_channel == NULL)
          {
           /* oops a dangling connection */

           fprintf(stderr,"0YSTEM INITIALIZATION ERROR: dangling");
           fprintf(stderr," channel in an instance of
                                    ptr->p_ident);
           fprintf(stderr,"channel number %d, index %d0,c_ptr->c_id,
                                        c_ptr->index_num);
          }
       }
    }


  /* join the ends of the process list into a loop */

  for (ptr = process_list; ptr->next != NULL; ptr = ptr->next);

  ptr->next =       process_list;


  return(process_list);

}

/*-------------------------------------------------------------*/
```

```
schedule(process_list)
/*
  This is the main driving routine.
*/

   struct  process_block   *process_list;


     {
      extern     int    signal_pending;
      extern int in_count, out_count;


      struct     channel_block    *c_ptr;
      struct     signal_block     *s_ptr, *get_signal();
      struct     process_block    *p_ptr, *p_ptr2;


      p_ptr2     =    process_list;
      p_ptr =    process_list;
      c_ptr  =    p_ptr->chan_list;
      signal_pending      = 0;

      while ( (in_count < 5) || (out_count <5))
           /* arbitrary stop condition */
        {
           if ((s_ptr = get_signal(&c_ptr, &p_ptr))
                      != NULL)
             {

           /* call the transition routine */

           (*(p_ptr->proc_ptr))(p_ptr,c_ptr,s_ptr);
             }

           /* move on to the next channel for the
             start of the next search */

           if (c_ptr->next != NULL)
             c_ptr = c_ptr->next;

           else
              {
              p_ptr = p_ptr->next;
              c_ptr = p_ptr->chan_list;
              }


        if ((strcmp(p_ptr2->p_ident,"timer_process") == 0) ||
             (strcmp(p_ptr2->p_ident,"user_process") == 0))
                (*(p_ptr2->proc_ptr))(p_ptr2,NULL,NULL);

        p_ptr2 = p_ptr2->next;
```

```
    }

  N_close(&conn);

  }

/*-------------------------------------------------------------*/


struct      signal_block      *get_signal(c_ptr,p_ptr)
/*
 This function finds a pending signal on a signal queue.
*/


   struct  channel_block     *(*c_ptr);
   struct  process_block     *(*p_ptr);


    {
     struct      signal_block      *s_ptr, *s_ptr2;
     int               found=   FALSE;
     int               n, temp;
     extern NET_CONN   conn;
     extern int        in_count, out_count;

     static int count;


     s_ptr2 = NULL;

     for (;!found; (*p_ptr) = (*p_ptr)->next,(*c_ptr)=(*p_ptr)->chan_list,
            s_ptr2 = NULL)
      {
       for (; (((*c_ptr) != NULL) && (!found));
            (*c_ptr)= (*c_ptr)->next)
        {
         if ((*c_ptr)->signal_list != NULL)
           {
            found = TRUE;
            if ((*c_ptr)->signal_list->next != NULL){
              for (s_ptr = (*c_ptr)->signal_list,
                  s_ptr2 = s_ptr->next;
                 s_ptr2->next != NULL;
              s_ptr = s_ptr2, s_ptr2 = s_ptr2->next);

              s_ptr->next = NULL;
              }
            else
              {
               s_ptr2               =  (*c_ptr)->signal_list;
               (*c_ptr)->signal_list =  NULL;
```

```
                }
               goto end;

              }

            if ((strcmp((*p_ptr)->p_ident,"net_proc") == 0) &&
                ((*c_ptr)->c_id == 3))
               {


               if ((n = N_receive(&conn, buf, buffsize)) <= 0)
                {
                if (n < 0)
                   {
                    printf(" read error %d0,n);
                    exit(1);
                   }
                }
               else
                  {
                    s_ptr2 = ALLOCATE(signal_block);
                    s_ptr2->signal_id = buf[0];
                    s_ptr2->lvars.network_channel.packet.id = buf[1];
                    s_ptr2->lvars.network_channel.packet.data = buf[2];
                    s_ptr2->lvars.network_channel.packet.seq = buf[3];

                    in_count++;

                    goto end;

                  }

               s_ptr2 = NULL;
               goto end;

               }

          }
       }
   end:
      return(s_ptr2);

     }

/*-------------------------------------------------------------*/
#include <sys/types.h>
#include <sys/timeb.h>

#include "listdefs.h"

struct      time_struct
  {
   int          flag;
```

```
    int         secs;
    int         millisecs;
  };



schedule_timeout(timer,time_value)
/*
 This routine places in the timer passed the value of present time +
 "time_value" which will be tested later by "timer_expired". The units
 of time_value are milliseconds.
*/

    struct time_struct  *timer;
    int                 time_value;
     {
      struct      timeb      *tp;


      ftime(tp);

      timer->flag      =   TRUE;

      timer->secs      =   tp->time;

      timer->millisecs=   tp->millitm;


      if ((time_value = time_value + tp->millitm) >= 1000)
        {
         timer->secs++;
         timer->millisecs    =   time_value - 1000;
        }

     }



/*-------------------------------------------------------------*/


int timer_expired(timer)
/*
 This predicate returns TRUE if the timer passed has expired and cancels
 it if it has


*/

    struct  time_struct     *timer;
```

```
    {
     struct      timeb      *tp;


     ftime(tp);


     if ((timer->flag == TRUE) &&
        (timer->secs < tp->time) &&
        (timer->millisecs < tp->millitm))

        return(TRUE);

     else return(FALSE);

    }

/*-------------------------------------------------------------*/


cancel_timer(timer)
/*
  This routine cancels the timer passed
*/

   struct  time_struct *timer;
     {
      timer->flag = FALSE;
     }
```

Appendix VI

Alternating Bit Implementation Code

```
#include <stdio.h>
#include <strings.h>
#include "listdefs.h"


/* type code */
typedef int data_type;
typedef int seq_type;
typedef int id_type;
typedef int timer_type;
typedef struct {
     id_type id;
     data_type data;
     seq_type seq;
}
ndata_type;
typedef struct {
     data_type msgdata;
     seq_type msgseq;
}
msg_type;
typedef struct {
     int flag;
     msg_type msg;
}
buffer_type;
typedef int int_type;
typedef struct {
     int flag;
     int secs;
     int millisecs;
}
time_struct[1];
typedef int request_type;
typedef int message_value_type;

/*-----------------------------------*/


/* signal block dcl */
```

```
struct signal_block {
      int signal_id;
      struct signal_block  *next;
      union {
            struct {
                  struct {
                        data_type udata;
                  }
                  receive_response;
            }
            u_receive_point;
            struct {
                  struct {
                        data_type udata;
                  }
                  send_request;
            }
            u_send_point;
            struct {
                  struct {
                        timer_type name;
                        int  time;
                  }
                  timer_request;
                  struct {
                        timer_type name;
                  }
                  timer_response;
            }
            s_access_point;
            struct {
                  struct {
                        id_type id;
                        data_type data;
                        seq_type seq;
                  }
                  data_request;
                  struct {
                        id_type id;
                        data_type data;
                        seq_type seq;
                  }
                  data_response;
            }
            n_access_point;
            struct {
                  struct {
                        id_type id;
                        data_type data;
                        seq_type seq;
                  }
                  packet;
            }
            network_channel;
```

```
        }
        lvars;
};

/*-_----------------------------*/


/* process dcl code */
struct process_block {
        struct process_block *next;
        char p_ident[MAX_IDENT_LENGTH+1];
        struct channel_block *chan_list;
        struct process_block *refinement;
        int    (*proc_ptr)();
        union {
                struct {
                        int  state;
                        seq_type send_seq;
                        seq_type recv_seq;
                        buffer_type send_buffer;
                        buffer_type recv_buffer;
                        msg_type p,  q;
                }
                s_alternating_process;
                struct {
                        timer_type index;
                        time_struct a_timer;
                }
                s_timer_process;
                struct {
                        int  user_number;
                        int  count;
                }
                s_user_process;
        }
        lvars;
};

/*-----------------------------------*/


/* procedure code */
struct process_block *ioab_ref(p_block) struct process_block *p_block;
{
        struct process_block *ioalternating_process();
        struct process_block *iotimer_process();
        struct process_block *ionet_proc();
        struct process_block *iouser_process();
        struct channel_block *add_channel_block(), *c_ptr;
        struct process_block *add_refinement_header(), *clean_up(),
                             *process_list;
        process_list = NULL;
        process_list =  ioalternating_process(process_list);
        process_list =  iotimer_process(process_list);
```

```
        process_list =  ionet_proc(process_list);
        process_list =  iouser_process(process_list,1);
        connect_ports(process_list,4, 1, 0, 1, 1, 0);
        connect_ports(process_list,4, 2, 0, 1, 2, 0);
        connect_ports(process_list,4, 3, 0, 2, 1, 0);
        connect_ports(process_list,4, 4, 0, 3, 1, 0);
        p_block = add_refinement_header(p_block,process_list);
        p_block->refinement = clean_up(p_block->refinement);
        return(p_block);
}

/*-----------------*/

struct process_block *ioalternating_process(process_list)

struct process_block *process_list;
{
        struct process_block *p_block, *add_process_block();
        extern int alternating_process();
        struct channel_block *add_channel_block(), *c_ptr;
        p_block = add_process_block(process_list,alternating_process,
                "alternating_process");
        p_block->chan_list = add_channel_block(p_block->chan_list,
                &c_ptr,TRUE,1,0);
        p_block->chan_list = add_channel_block(p_block->chan_list,
                &c_ptr,TRUE,2,0);
        p_block->chan_list = add_channel_block(p_block->chan_list,
                &c_ptr,TRUE,3,0);
        p_block->chan_list = add_channel_block(p_block->chan_list,
                &c_ptr,TRUE,4,0);
        p_block->lvars.s_alternating_process.state = 1;
        p_block->lvars.s_alternating_process.send_seq = 0;
        p_block->lvars.s_alternating_process.recv_seq = 0;
        p_block->lvars.s_alternating_process.send_buffer.flag = 0;
        p_block->lvars.s_alternating_process.recv_buffer.flag = 0;
        return(p_block);
}

/*-----------------*/

alternating_process(p_block,channel,signal)

struct process_block *p_block;
struct channel_block *channel;
struct signal_block *signal;
{
        struct signal_block *s_ptr;
        if ((channel != NULL)  &&
           ((p_block->lvars.s_alternating_process.state == 1)))
          if ((channel->c_id == 1) && (signal->signal_id == 2)) {
           if (!(p_block->lvars.s_alternating_process.send_buffer.flag)){
              p_block->lvars.s_alternating_process.state = 0 ;
              {
              p_block->lvars.s_alternating_process.send_buffer.flag = 1;
```

```
        p_block->lvars.s_alternating_process.send_buffer.msg.msgdata =
              signal->lvars.u_send_point.send_request.udata;
        p_block->lvars.s_alternating_process.send_buffer.msg.msgseq =
              p_block->lvars.s_alternating_process.send_seq;

        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 5;
        s_ptr->lvars.n_access_point.data_request.id = 0;
        s_ptr->lvars.n_access_point.data_request.data =
              signal->lvars.u_send_point.send_request.udata;
        s_ptr->lvars.n_access_point.data_request.seq =
              p_block->lvars.s_alternating_process.send_seq;
        out(p_block->chan_list,s_ptr,3,0);
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 3;
        s_ptr->lvars.s_access_point.timer_request.name = 0;
        s_ptr->lvars.s_access_point.timer_request.time = 100;
        out(p_block->chan_list,s_ptr,4,0);
        }
     goto dispose;
}
}
if ((channel != NULL)  &&
    ((p_block->lvars.s_alternating_process.state == 1 )  ||
    (p_block->lvars.s_alternating_process.state == 0)))
   if ((channel->c_id == 2) && (signal->signal_id == 0)) {
        if (p_block->lvars.s_alternating_process.recv_buffer.flag){
        {
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 1;
        s_ptr->lvars.u_receive_point.receive_response.udata =
         p_block->lvars.s_alternating_process.recv_buffer.msg.msgdata;
        out(p_block->chan_list,s_ptr,2,0);
        p_block->lvars.s_alternating_process.recv_buffer.flag = 0;
        }
        goto dispose;
   }
}
if ((channel != NULL)  &&
    ((p_block->lvars.s_alternating_process.state == 0)))
  if ((channel->c_id == 4) && (signal->signal_id == 4)) {
        if ((signal->lvars.s_access_point.timer_response.name == 0)){
        p_block->lvars.s_alternating_process.state = 0 ;
        {
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 5;
        s_ptr->lvars.n_access_point.data_request.id = 0;
        s_ptr->lvars.n_access_point.data_request.data =
          p_block->lvars.s_alternating_process.send_buffer.msg.msgdata;
        s_ptr->lvars.n_access_point.data_request.seq =
          p_block->lvars.s_alternating_process.send_buffer.msg.msgseq;
        out(p_block->chan_list,s_ptr,3,0);
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 3;
```

```
        s_ptr->lvars.s_access_point.timer_request.name = 0;
        s_ptr->lvars.s_access_point.timer_request.time = 100;
        out(p_block->chan_list,s_ptr,4,0);
        }
        goto dispose;
}
}
if ((channel != NULL)  &&
    ((p_block->lvars.s_alternating_process.state == 1)))
  if ((channel->c_id == 4) && (signal->signal_id == 4)) {
        if ((signal->lvars.s_access_point.timer_response.name == 0)){
          p_block->lvars.s_alternating_process.state = 1 ;
        {
        p_block->lvars.s_alternating_process.state =
                p_block->lvars.s_alternating_process.state;
        }
        goto dispose;
        }
}
if ((channel != NULL)  &&
    ((p_block->lvars.s_alternating_process.state == 0)))
  if ((channel->c_id == 3) && (signal->signal_id == 6)) {
        if (ack_ok(signal->lvars.n_access_point.data_response.id,
           signal->lvars.n_access_point.data_response.seq,
           p_block->lvars.s_alternating_process.send_seq)){
        p_block->lvars.s_alternating_process.state = 1 ;
        {
        p_block->lvars.s_alternating_process.send_buffer.flag = 0;
        p_block->lvars.s_alternating_process.send_seq =
         (((p_block->lvars.s_alternating_process.send_seq + 1)) % 2);
        }
        goto dispose;
}
}
if ((channel != NULL)  &&
    ((p_block->lvars.s_alternating_process.state == 1 ) ||
    (p_block->lvars.s_alternating_process.state == 0)))
  if ((channel->c_id == 3) && (signal->signal_id == 6)) {
    if ((signal->lvars.n_access_point.data_response.id == 0)){
        {
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 5;
        s_ptr->lvars.n_access_point.data_request.id = 1;
        s_ptr->lvars.n_access_point.data_request.data =
                signal->lvars.n_access_point.data_response.data;
        s_ptr->lvars.n_access_point.data_request.seq =
                signal->lvars.n_access_point.data_response.seq;
        out(p_block->chan_list,s_ptr,3,0);

        if((signal->lvars.n_access_point.data_response.seq ==
          p_block->lvars.s_alternating_process.recv_seq)) {
        {
        p_block->lvars.s_alternating_process.recv_buffer.flag = 1;
        p_block->lvars.s_alternating_process.recv_buffer.msg.msgdata =
```

```
                signal->lvars.n_access_point.data_response.data;
            p_block->lvars.s_alternating_process.recv_buffer.msg.msgseq =
                signal->lvars.n_access_point.data_response.seq;
            p_block->lvars.s_alternating_process.recv_seq =
             (((p_block->lvars.s_alternating_process.recv_seq + 1)) % 2);
            }
        }
        }
        goto dispose;
        }
        }

        if (channel != NULL) {
            requeue(channel,signal);
            signal=NULL;
        }
dispose:
        free(signal);
}

/*--------------------*/

int  ack_ok(id,seq,send_seq)id_type id;
seq_type seq,  send_seq;
{
        int  rtv_ack_ok;
        {
            rtv_ack_ok = (((id == 1)) && ((seq == send_seq)));
        }
        return(rtv_ack_ok);
}

/*--------------------*/

struct process_block *iotimer_process(process_list)
struct process_block *process_list;
{
struct process_block *p_block, *add_process_block();
extern int timer_process();
struct channel_block *add_channel_block(), *c_ptr;

p_block = add_process_block(process_list,timer_process,"timer_process");
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,TRUE,1,0);
for (p_block->lvars.s_timer_process.index = 0;
        p_block->lvars.s_timer_process.index++ <= 0;){
{
p_block->lvars.s_timer_process.a_timer[
  p_block->lvars.s_timer_process.index].flag = 0;
p_block->lvars.s_timer_process.a_timer[
  p_block->lvars.s_timer_process.index].secs = 0;
p_block->lvars.s_timer_process.a_timer[
  p_block->lvars.s_timer_process.index].millisecs = 0;
}
}
```

```
return(p_block);
}

/*-------------------*/

timer_process(p_block,channel,signal)

struct process_block *p_block;
struct channel_block *channel;
struct signal_block *signal;
{
      struct signal_block *s_ptr;

      if ((channel != NULL) )
       if ((channel->c_id == 1) && (signal->signal_id == 3)) {
       {
       {
       schedule_timeout(&p_block->lvars.s_timer_process.a_timer[
            signal->lvars.s_access_point.timer_request.name],
            signal->lvars.s_access_point.timer_request.time);
       }
      goto dispose;
            }
      }

      if (channel != NULL) {
            requeue(channel,signal);
            signal=NULL;
      }

      {
      timer_type index;
      index = random_select(0,0 );
      if (timer_expired(&p_block->lvars.s_timer_process.a_timer[index])) {
       {
        p_block->lvars.s_timer_process.a_timer[index].flag = 0;
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 4;
        s_ptr->lvars.s_access_point.timer_response.name = index;
        out(p_block->chan_list,s_ptr,1,0);
       }
      goto dispose;
      }
      }

dispose:
      free(signal);
}

/*-------------------*/

struct process_block *ionet_proc(process_list)

struct process_block *process_list;
```

```
{
struct process_block *p_block, *add_process_block();
extern int net_proc();
struct channel_block *add_channel_block(), *c_ptr;
p_block = add_process_block(process_list,net_proc,"net_proc");
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,TRUE,1,0);
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,FALSE,2,0);
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,TRUE,3,0);
return(p_block);
}

/*-----------------*/

net_proc(p_block,channel,signal)

struct process_block *p_block;
struct channel_block *channel;
struct signal_block *signal;
{
struct signal_block *s_ptr;
if ((channel != NULL) )
  if ((channel->c_id == 1) && (signal->signal_id == 5)) {
   {
   {
   s_ptr = ALLOCATE(signal_block);
   s_ptr->signal_id = 7;
   s_ptr->lvars.network_channel.packet.id =
        signal->lvars.n_access_point.data_request.id;
   s_ptr->lvars.network_channel.packet.data =
        signal->lvars.n_access_point.data_request.data;
   s_ptr->lvars.network_channel.packet.seq =
        signal->lvars.n_access_point.data_request.seq;
   out(p_block->chan_list,s_ptr,2,0);
   }
goto dispose;
}
}
if ((channel != NULL) )
  if ((channel->c_id == 3) && (signal->signal_id == 7)) {
  {
  {
  s_ptr = ALLOCATE(signal_block);
  s_ptr->signal_id = 6;
  s_ptr->lvars.n_access_point.data_response.id = signal->lvars.network_channel.packet.id;
  s_ptr->lvars.n_access_point.data_response.data = signal->lvars.network_channel.packet.data;
  s_ptr->lvars.n_access_point.data_response.seq = signal->lvars.network_channel.packet.seq;
  out(p_block->chan_list,s_ptr,1,0);
  }
goto dispose;
}
}

if (channel != NULL) {
      requeue(channel,signal);
```

```
        signal=NULL;
}

dispose:
        free(signal);
}

/*-----------------*/

struct process_block *iouser_process(process_list,number)

struct process_block *process_list;
int  number;
{
struct process_block *p_block, *add_process_block();
extern int user_process();
struct channel_block *add_channel_block(), *c_ptr;
p_block = add_process_block(process_list,user_process,"user_process");
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,TRUE,1,0);
p_block->chan_list = add_channel_block(p_block->chan_list,&c_ptr,TRUE,2,0);
p_block->lvars.s_user_process.user_number = number;
p_block->lvars.s_user_process.count = 0;
return(p_block);
}

/*-----------------*/

user_process(p_block,channel,signal)
struct process_block *p_block;
struct channel_block *channel;
struct signal_block *signal;
{
struct signal_block *s_ptr;
if ((channel != NULL) )
  if ((channel->c_id == 2) && (signal->signal_id == 1)) {
  {
  {
  printf("User ");
  printf("%d",p_block->lvars.s_user_process.user_number);
  printf(" received the message ");
  printf("%d",signal->lvars.u_receive_point.receive_response.udata);
  printf("0);
  }
 goto dispose;
}
}

if (channel != NULL) {
        requeue(channel,signal);
        signal=NULL;
}

{
request_type request;
```

```
message_value_type message;
request = random_select(0,1 );
message = random_select(1,9 );
if ((request == 1)) {
        {
        s_ptr = ALLOCATE(signal_block);
        s_ptr->signal_id = 0;
        out(p_block->chan_list,s_ptr,2,0);
        }
goto dispose;
}
}

{
request_type request;
message_value_type message;
request = random_select(0,1 );
message = random_select(1,9 );
if ((((request == 0)) &&
   ((p_block->lvars.s_user_process.count < 5)))) {
  {
  printf("User ");
  printf("%d",p_block->lvars.s_user_process.user_number);
  printf(" sent the message ");
  printf("%d",message);
  printf("0);
  p_block->lvars.s_user_process.count =
        (p_block->lvars.s_user_process.count + 1);
  s_ptr = ALLOCATE(signal_block);
  s_ptr->signal_id = 2;
  s_ptr->lvars.u_send_point.send_request.udata = message;
  out(p_block->chan_list,s_ptr,1,0);
 }
 goto dispose;
}
}

dispose:
      free(signal);
}

/*--------------------*/


#include "fdtutil.c"
```

Appendix VII

Running The Compiler

To run the compiler is simple. Simply enter:

- fdt <inputfile >fdt.c

The outputfile *fdt.c* will contain the generated *C* code which is compiled and linked with the utility routines with the following command:

- make

This runs the *make* command which knows how to "make" programs, see the manual for further details.