View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library

Presented at ASEE National Convention, Rochester NY, 6/83

UTEC-83-030

TRANSPARENT INTERFACE BETWEEN SOFTWARE AND HARDWARE VERSIONS OF ADA COMPILATION UNITS

by

Elliott I. Organick, Michael P. Maloney,
Dan Klass and Gary Lindstrom

Department of Computer Science
University of Utah
Salt Lake City Utah

April 1983

Sponsored by

Defense Advanced Research Projects Agency (DoD)
ARPA Order No. 4305

Under Contract No. MDA 903-81-C-0411, issued by
Defense Supply Service - Washington, Washington DC 20310

Table of Contents

1. Introduction 1
2. High-level or functional test of hardware components 1
3. Test Strategy and Test Support Details 2
4. Efficiency Issues Concerning the Adaptor 4
5. Application of the Test Strategy to the RIP_Manager Chip 5
6. Conclusion 7
Appendix: The RIP_Interface Package 10

List of Figures

Figure 5-1: Read_Init_Parameters Software Environment 5
Figure 5-2: The RIP_Interface package 6
Figure 5-3: iAPX 432 Chip Tester System for RIP_Manager 7
Figure 5-4: The Communication Interface to the RIP_Manager Chip 8

1. Introduction

The Ada-to-Silicon project at Utah is developing a methodology (and associated software and hardware) for the high level testing of Ada compilation units that are represented as hardware components (circuitry). There are two motivations for this research:

1. The initial motivation is to facilitate migration of selected program components (in Ada) from software to hardware by providing a means for interfacing parts of Ada programs that execute on general purpose hosts with parts that are represented directly in special purpose hardware (e.g. VLSI). A companion aim is to investigate and recommend a methodology of Ada programming such that a decision to migrate a component to hardware minimizes the reprogramming effort of the non-migrated part. Ideally, the interface between the "soft" and "hard" parts should be totally transparent, so that no change whatsoever is required for the soft part. (This ideal, though it may eventually be met, is not quite reached in the methodology reported below.)

2. The second motivation is related to test-bedding in a more general sense. In designing any subsystem S for hardware implementation, it is essential to simulate and test it to confirm that its functional specifications are met. Suppose S is specified in Ada; more particularly, suppose the interface between S and its environment (S's input/output characteristics) are specified. In that case, a driver program specified and implemented in Ada can be linked first to the software version of S to simulate its behavior, and later to the hardware version of S to test it. Once tested in this way, S can be disconnected from its driver and incorporated into the system for which it was initially designed. The transparent interface logic required to link soft and hard parts of an Ada program also serves for testing S.

In view of the two motivations just described, we now refer to the interface methodology, the host system, and software required to utilize it, as our "environmental test bed for VLSI". A first version should be completed and in place in time for use in testing components of a new applicative multiprocessing system under development at Utah and known as Rediflow (reduction and data flow) architecture.

2. High-level or functional test of hardware components

By the high level test of a component B we mean the following. Let P be an Ada program, some compilation units of which are to be implemented directly in hardware. The component B has semantics equivalent to some Ada package that contains one or more embedded tasks, each of which accepts or issues entry calls to/from other units of P.

The test of B hinges on demonstrating that program P is functionally unchanged (timing constraints aside), no matter if it executes by interacting

with hardware component B or by interacting instead with its (soft) Ada equivalent. If, for example, the objective is to test the fast packet switch of Rediflow, we would model it as an Ada package and design an Ada program P that serves as a driver. That driver would be used to exercise the software model of the packet switch (thus simulating it at the Ada level) and then, after replacing the model of the switch with its hardware equivalent, to exercise the corresponding hardware. Getting identical results when executing P using the software and hardware versions of B constitutes a positive test of the circuit under development.

As will be further explained in this paper, hardware considerations currently dictate that each entry call to a task embedded in B or from B must first be mapped to an equivalent pair of Send and Receive operations. A Send operation transmits a message consisting of the in-bound parameter values; the Receive operation obtains a message consisting of the out-bound parameter values of the entry call. In the case of no out-bound values, the receive message serves as an acknowledgement indicating rendezvous completion. Messages are transmitted though typed ports which are message queue instances of specified types [1]. (Tasks that communicate with their environment exclusively via typed ports can also be viewed as Kahn Processes [2].)

The connection of a hardware version of B with the host that executes the remainder of program P (e.g., a driver program) is accomplished through suitably-defined asynchronous hardware and software interface, or "adaptor". This adaptor is, by design, transparent to program P and the packet switch component. Our work on the test bed has led us to the design of a particular style of adaptor, which will be used first in testing the Read_Init_Parameters task (of the INM_OUT in the Internet Protocol [3]) and its containing package RIP_Manager. The result of this work is expected to be a set of ideas on how to design adaptors for a wide class of applications including those that arise in building and testing key parts of the Rediflow architecture.

3. Test Strategy and Test Support Details

The scenario for using the adaptor and a sketch of the adaptor structure is described below. As a given, we choose the iAPX 432 System [4] (now installed and operating at Utah, or faster follow-on versions when available) as the host which executes program P.[1] The effective speed of this host may lag those of the hardware parts we attach though the adaptor. However, since the interface is asynchronous, such speed differentials do not affect our test strategy.

---

[1] It is conceivable that other object-based architectures that support Ada might also prove suitable, but equally suitable alternative hosts are not now on the horizon.

In essence, the strategy is as follows:

1. Each entry call to a task in B from a requestor in the rest of P is converted into a two-way, message-based communication relationship using typed request and response ports (in the iAPX 432 sense and, equivalently, a Kahn Process sense in the programmatic sense). Messages sent to typed request ports are received by the adaptor and forwarded to B using appropriate hardware (asynchronous) protocols. Response (acknowledge) messages transmitted by B to complete the equivalent rendezvous reach the unit in P that has issued the entry call in the following way. The adaptor Sends the response from B as a message to the appropriate response port, to which the caller issues a Receive call.

   Inter-task communication to be conducted entirely within B is converted to appropriate hardware asynchronous message-based communication (but we are not concerned with such internal communication here.)

2. Each entry call to a server task in P from a requestor in B is also converted by the adaptor into a two-way, message-based communication relationship using typed ports. This protocol is the same as above, but with the direction of the communication reversed. Thus, an entry call from B to P is implemented in hardware using a Request/Acknowledge protocol that interfaces with the same transparent software/hardware interface (adaptor). Messages flowing through this interface are then converted to i432 Send/Receive operations on typed message ports in the i432 object space. (One can assure that no changes are required in the encoding of the server task in P if request messages from B are directed to an auxiliary task nested within the "shadow package" described next. The auxiliary task would receive these messages and convert them into entry calls to the appropriate target(s) in P.) We refer to this task as an Entry_Call_Forwarder task.

3. An image (or "shadow") version of the package B is retained as a "placeholder" of B in P. The shadow version is used to hide or minimize the conversions cited above. In essence, entry calls to B are replaced by procedure calls to subprograms in the shadow, which in turn convert these calls to Send/Receive pairs constituting a protocol that is the equivalent of the replaced Rendezvous. (Note that the extra steps of indirection implied by operations of the shadow package, which functions as a software communications switch and which may eventually be generated by a smart compiler, is not expected to add materially to the overhead of the test-bed system as a whole.)

   The Send/Receive pairs are implemented in part using i432 hardware and software and in part (currently) by conventional (8086-based) software/hardware associated with the use of the attached multibus, a peripherals interface, and its controls. All of this software and hardware, which is what constitutes the adaptor, is transparent. The hardware portion is off-the-shelf and the software portion requires

only a small amount of yet-to-be-written special, but well-understood, software. Later versions of the 432 may well have more direct I/O facilities than those currently provided. If so, the hardware/software requirements mentioned here would be significantly simplified. That is, it may be possible to eliminate the attached processor and multibus and go directly from an I/O processor of the 432 to the hardware version of C. (Whether this will come about by use of commercially-provided or custom-designed I/O processors is not yet clear.)

The substitution of a software task by a "shadow" package present following two implications:

1. What was previously an entry call can now be represented as a procedure call. Since the syntax of procedure calls and entry calls is identical, only recompilation of surrounding software elements is required in most cases. The semantic change propagated by substituting a procedure call for an entry call eliminates the use of timed and conditional entry calls to the hardware task.[2]

2. In the transition from an entry call to a procedure call, some useful features of entry calls such as arbitration between caller and entry queue management are lost. When these features are needed in particular cases, they can be easily provided explicitly in the implementation of the shadow package.

4. Efficiency Issues Concerning the Adaptor

The "impedance" presented by the transparent interface may be consid relative to the raw speed capability of the component under test. Howeve purpose of the particular interface reported here is primarily for t hardware components produced from an Ada-to-Silicon transformation method. In well-planned applications (where B is viewed as an integral part of P) an elaborate interface would only be justified where its impedance c tolerated. (In any event, use of expected replacement versions of th architecture will significantly reduce this impedance.)

In the case of the Rediflow architecture, for example, once the fast s is tested, it would probably be integrated with other parts of the archite that do not require such an elaborate interface for testing. (For an e Rediflow array, it may not be necessary to maintain or to exercise any interface during normal operations; use of such interfaces would be confine execution of input/output (including file) processing, or when perfor diagnostic, reconfiguration, or alarm processing.

---

[2]Although implementation of timed and conditional entry calls entirely software or entirely in hardware may be possible, it is unclear how th constructs can be implemented when they cross the software/hardware boundary

5. Application of the Test Strategy to the RIP_Manager Chip

Here we present our intended interface to an actual piece of hardware which demonstrates the migration of a particular Ada package (RIP_Manager) and its contained task (Read_Init_Parameters) from software. The original environment from which the RIP_Manager package was extracted (Figure 5-1) includes three other packages and their embedded tasks [3, 5]

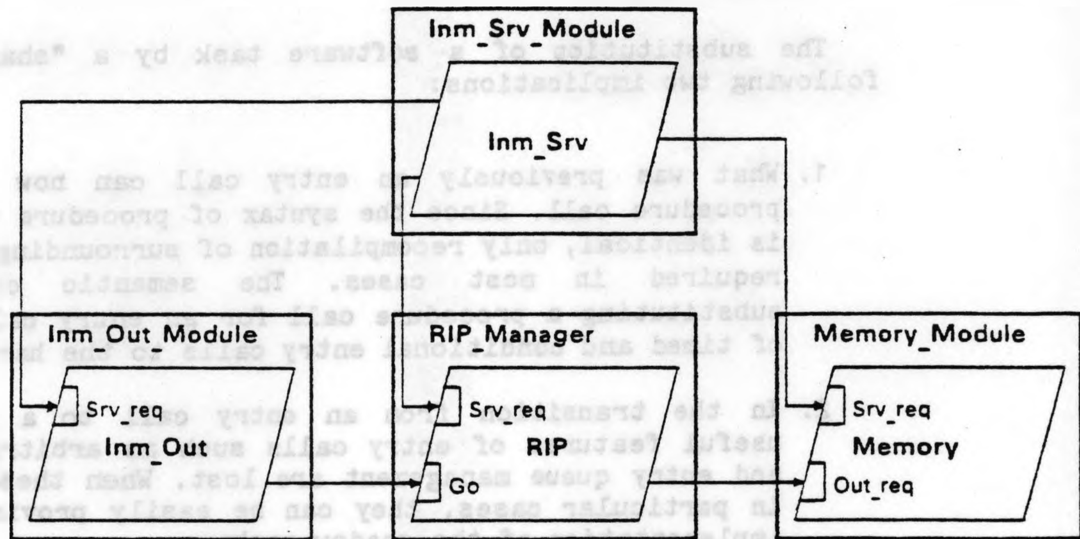

Figure 5-1: Read_Init_Parameters Software Environment

The task Read_Init_Parameters receives entry calls from tasks within packages Inm_Out_Module and Inm_Srv_Module. Read_Init_Parameters issues an entry call to a task within Memory_Module.

The first of the transformations requires the replacement of all rendezvous communication to/from Read_Init_Parameters with port-based communication (Figure 5-2). We then enclose RIP_Manager with an interface package, RIP_Interface, which owns the communication ports and provides public procedures in place of entry calls to the Read_Init_Parameters task. Request and reply ports are provided for each procedure of RIP_Interface (corresponding to each entry of the Read_Init_Parameters task). An Entry_Call_Forwarder task along with another pair of (request, reply) ports is provided for the entry of the Memory task of Memory_Module that is called by Read_Init_Parameters. An "entry call" to Read_Init_Parameters is now represented as a call to the appropriate procedure of RIP_Interface, which then Sends inbound parameters to a request port, and Receives outbound parameters from a response port. The calling task is suspended pending service from Read_Init_Parameters because the "entry procedure" does not return until the outbound messages are actually Received from the response port.

An entry call from Read_Init_Parameters to the Memory task is now performed by the Entry_Call_Forwarder task. This task waits for a command (containing inbound parameters) at the Mem_Req port to perform a rendezvous with Memory. When the rendezvous is terminated, the entry_call_forwarder task will Send the

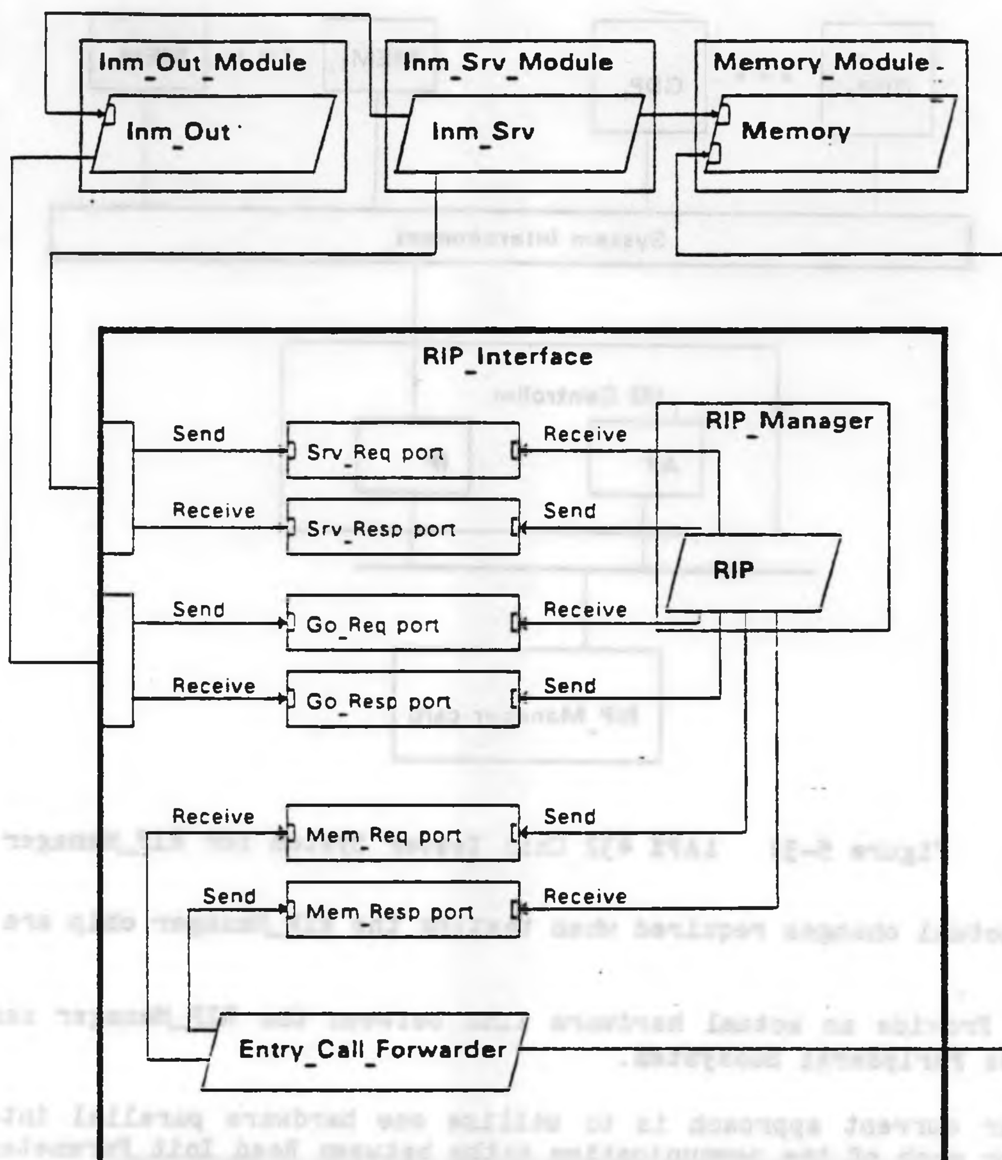

Figure 5-2: The RIP_Interface package

outbound parameters to the Mem_Resp port. The Ada code for this RIP_Interface package is in the Appendix.

The environment in which the RIP_Manager chip will be tested, as illustrated in Figure 5-3, consists of an Intel 432/600 System. The RIP_Manager card interfaces to a Peripheral Subsystem of the i432 managed by an I/O Controller. This I/O Controller consists of an i432 Interface Processor combined with a Multibus-compatible Attached Processor (8086 based, in our case). It provides port-based communication facilities to RIP_Manager. The RIP_Interface package (without RIP_Manager) can now be thought of as a "communication interface" to the RIP_Manager chip (Figure 5-4). The port operations needed to communicate with the software modules are now performed for the RIP_Manager chip by the I/O Controller.

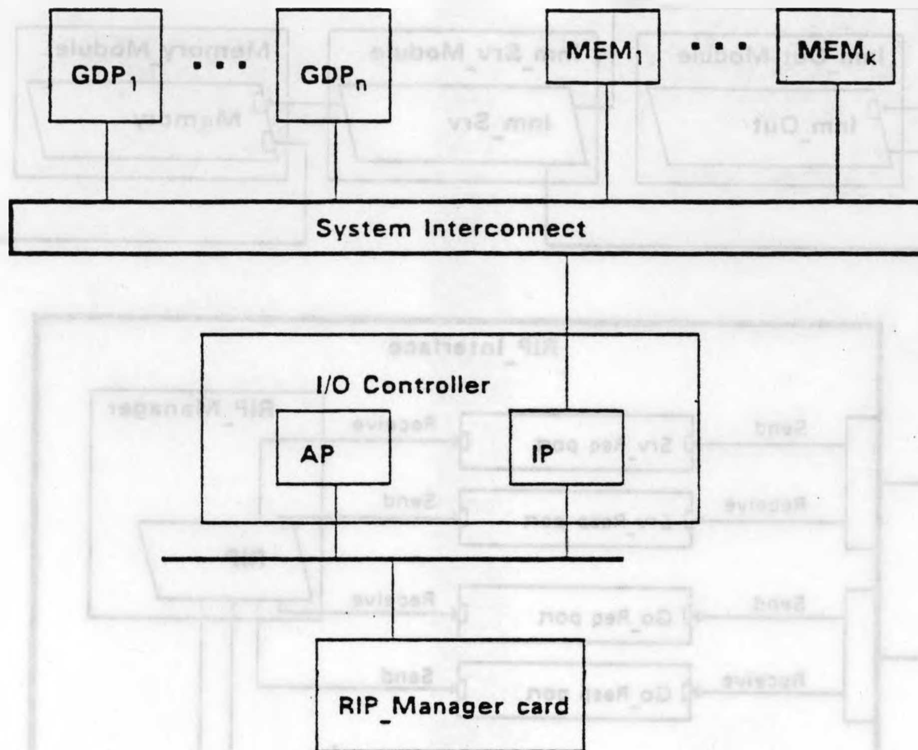

Figure 5-3: iAPX 432 Chip Tester System for RIP_Manager

The actual changes required when testing the RIP_Manager chip are then:

1. Provide an actual hardware link between the RIP_Manager card and the Peripheral Subsystem.

   Our current approach is to utilize one hardware parallel interface for each of the communication paths between Read_Init_Parameters and the software tasks.

2. Provide Peripheral Subsystem software to (1) utilize the parallel interfaces and (2) provide the RIP_Manager chip with message-based communication facilities.

3. Eliminate the software RIP_Manager from RIP_Interface. The Send and Receive operations provided by the Peripheral Subsystem Software (and associated hardware) allow the direct replacement of the software RIP_Manager package by the RIP_Manager chip.

6. Conclusion

In providing an interface to the hardware version of the RIP_Manager package we replaced rendezvous communication to the Read_Init_Parameters task with message-based communication using typed ports. This message based communication was made transparent to the other software packages by the

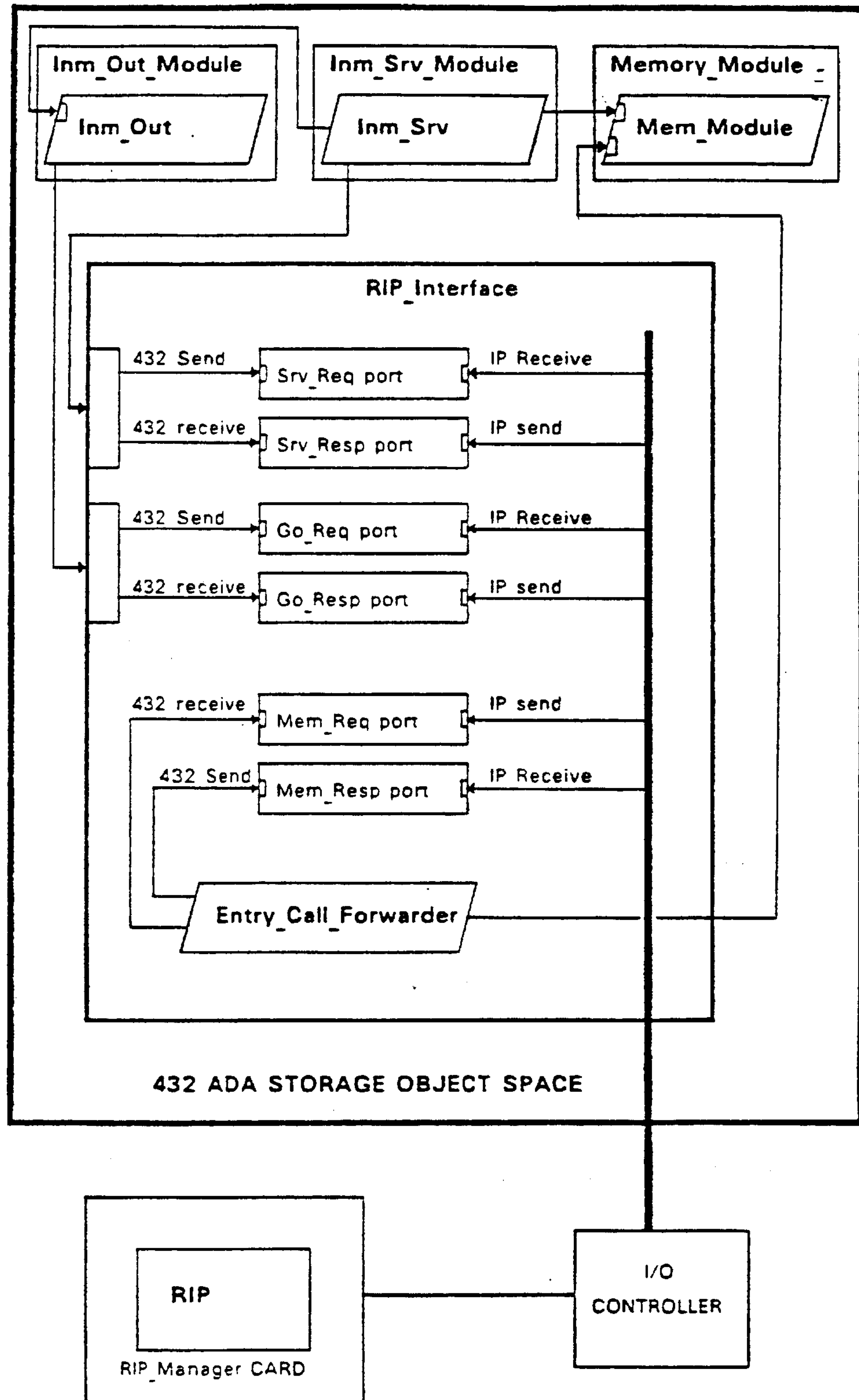

Figure 5-4: The Communication Interface to the RIP_Manager Chip

RIP_Interface package. The public "entry" procedures of RIP_Interface perform the communication necessary in implementing an entry call to the Read_Init_Parameters task. The Entry_Call_Forwarder task of RIP_Interface performs the entry calls made by the Read_Init_Parameters task to the Memory task. By providing this RIP_Interface package we minimize the changes to the other packages, Inm_Out_Module, Inm_Srv_Module and Memory_Module. The RIP_Interface package also allows us to test the port-based communication to/from the Read_Init_Parameters task of RIP_Manager. The replacement of the RIP_Manager package by an integrated circuit involves the following three steps:

1. The removal of the RIP_Manager package from the RIP_Interface package.

2. The construction of a physical connection to the RIP_Manager chip.

3. The provision for port-based communication facilities to the RIP_Manager chip.

Of these three steps, the first involves only the recompilation of the body of the RIP_Interface package with the RIP_Manager package now removed. The second involves the use of three hardware parallel interfaces to a Peripheral Subsystem of the Intel 432. The third includes the utilization and construction of appropriate Peripheral Subsystem software to provide the RIP_Manager chip with Send and Receive operations.

The hardware and software requirements for steps two and three will be detailed in a subsequent report. These three steps for achieving a transparent software/hardware interface are believed to widely applicable. The "proof" of this approach must await the demonstration of this example case, which we expect to complete and make operational later this year.

Appendix: The RIP_Interface Package

```
pragma environment("iosd.mse","oiod.mse",
                   "oiiod.mse","mem.mse",
                   "$IMAX/TYPORT.MLE");

with Typed_Ports,Inm_Out_Defs,In_Out_Srv_Defs,Inm_In_Out_Defs,Memory_Module;
use  Inm_Out_Defs,Memory_Module,In_Out_Srv_Defs,Inm_In_Out_Defs;


package RIP_Interface is

  procedure Go(
    init_num_formal:      bit3;
    response        : out out_response);
   --
   -- Function:
   --   Gets init_num address chunks from INM_SRV and ships them over to
   --   the associated Memory module, forming the base address of the
   --   storage block containing the initialization parameters; then
   --   gets the initialization parameters from the Memory module.
   --   Sets out_response to either send_ok if successful or to
   --   bad_srv_command if unsuccessful. (Can be unsuccessful if required
   --   tos table size exceeds available local space.)

  procedure Srv_req(
    server_command_datum:      srv_command_type;
    response_to_server:   out out_response);
   --
   -- Function:
   --   This entry receives commands from the INM_SRV module.
   --   Note that task Inm_Out has an identical entry.

end RIP_Interface;


package body RIP_Interface is

  -- Types used for passing messages to and from the Read_Init_Parameters task

  type memory_out_request_rep is
   record
    request_type_formal:      memory_request_type;
    chunk_of_address_formal:  chunk_of_address_type;
    octet_formal:             octet_type;
   end record;

  type memory_out_request_mess  is access memory_out_request_rep;
  type memory_out_response_mess is access octet_type;
  type go_request_mess          is access bit3;
  type go_response_mess         is access out_response;
  type srv_req_request_mess     is access srv_command_type;
```

```
type srv_req_response_mess     is access out_response;

-- Typed port package instantiation

package Memory_Out_Request_Port_Def is
  new Typed_Ports.Simple_Port_Def(memory_out_request_mess);
  -- Memory_Out_Request_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- memory_out_request_mess.

package Memory_Out_Response_Port_Def is
  new Typed_Ports.Simple_Port_Def(memory_out_response_mess);
  -- Memory_Out_Response_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- memory_out_response_mess.

package Go_Request_Port_Def is
  new Typed_Ports.Simple_Port_Def(go_request_mess);
  -- Go_Request_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- go_request_mess.
package Go_Response_Port_Def is
  new Typed_Ports.Simple_Port_Def(go_response_mess);
  -- Go_Response_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- go_response_mess.

package Srv_Req_Request_Port_Def is
  new Typed_Ports.Simple_Port_Def(srv_req_request_mess);
  -- Srv_Req_Request_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- srv_req_request_mess.

package Srv_Req_Response_Port_Def is
  new Typed_Ports.Simple_Port_Def(srv_req_response_mess);
  -- Srv_Req_Response_Port_Def.user_port is an access type consisting of
  -- accesses for ports that can only handle messages of type
  -- srv_req_response_mess.

-- renaming of the port packages user_port definitions

subtype memory_out_request_port_type
                              is Memory_Out_Request_Port_Def.user_port;
subtype memory_out_response_port_type
                              is Memory_Out_Response_Port_Def.user_port;
subtype go_request_port_type        is Go_Request_Port_Def.user_port;
subtype go_response_port_type       is Go_Response_Port_Def.user_port;
subtype srv_req_request_port_type   is Srv_Req_Request_Port_Def.user_port;
subtype srv_req_response_port_type  is Srv_Req_Response_Port_Def.user_port;
```

```
-- create instances of the ports

memory_out_request_port :  memory_out_request_port_type;
memory_out_response_port:  memory_out_response_port_type;
go_request_port          :  go_request_port_type;
go_response_port         :  go_response_port_type;
srv_req_request_port     :  srv_req_request_port_type;
srv_req_response_port    :  srv_req_response_port_type;


-- procedures used for communication with the RIP task

procedure Go(
    init_num_formal:     bit3;
    response         : out out_response)
is
  request_message  :  go_request_mess  := new bit3;
  response_message :  go_response_mess;
begin
  request_message.all := init_num_formal;
  Go_Request_Port_Def.send(go_request_port,request_message);
  Go_Response_Port_Def.receive(go_response_port,response_message);
  response            := response_message.all;
end Go;

procedure Srv_req(
    server_command_datum:     srv_command_type;
    response_to_server  : out out_response)
is
  request_message  :  srv_req_request_mess  := new srv_command_type;
  response_message :  srv_req_response_mess;
begin
  request_message.all := server_command_datum;
  Srv_Req_Request_Port_Def.send(srv_req_request_port,request_message);
  Srv_Req_Response_Port_Def.receive(srv_req_response_port,response_message);
  response_to_server  := response_message.all;
end Srv_req;

-- Renamed task entry:
---------------------
   -- The package Memory_Module containing the task Memory holds
   -- to-be-sent datagrams as well as initialization parameters
   -- needed by INM_OUT.
```

```
procedure Memory_out_request(
    request_type_formal    :        memory_request_type;
                                 -- Load_address or receive_datum_octet
                                 -- or load_datum_octet.
    chunk_of_address_formal:        chunk_of_address_type;
                                 -- Don't care when request_type_formal
                                 -- receive_datum_octet.
    octet_formal           : in out octet_type)
                                 -- Don't care when load_address.
  renames Memory.Out_request;

-- The following task rendezvous with the Memory_out_request entry

task Entry_Call_Forwarder_Task;
task body Entry_Call_Forwarder_Task is
  request_message  :  memory_out_request_mess;
  response_message :  memory_out_response_mess := new octet_type;
begin
  Memory_Out_Request_Port_Def.
                        receive(memory_out_request_port,request_message);
  Memory_out_request(
     request_type_formal     => request_message.request_type_formal,
     chunk_of_address_formal => request_message.chunk_of_address_formal,
     octet_formal            => request_message.octet_formal);
  response_message.all := request_message.octet_formal;
  Memory_Out_Response_Port_Def.
                        send(memory_out_response_port,response_message);
end Entry_Call_Forwarder_Task;


package RIP_Manager is

  task Read_Init_Parameters;

end RIP_Manager;

package body RIP_Manager is

  task body Read_Init_Parameters is

    -------------------
    -- The following initialization variables were originally located in the
    -- package Inm_Out_Module and are now located in the
    -- task body of Read_Init_Parameters.
    --Variables to hold initialization parameter values:
```

```
lnm_max_packet:                         two_octet_record;
                                        -- Largest size packet
                                        -- for the local net.
                                        -- Represented as a pair of
                                        -- octets and also used
                                        -- as a 16-bit integer after
                                        -- applying Unchecked_
                                        -- conversion.

lnm_address_length:                     octet_type;
                                        -- Used in Read_in_header.

lnm_time_out:                           two_octet_record;
                                        -- Waiting time at LN.
                                        -- Represented as a pair of
                                        -- octets and also used
                                        -- as a 16-bit integer after
                                        -- applying Unchecked_
                                        -- conversion.

ack_type:                               octet_type;
                                        -- Early/late.

local_net_type_of_service_table_row_size: octet_type;
number_of_local_net_types_of_service    : octet_type;

-- Tos array:
tos_table: octet_buffer_type(0 .. max_tos_table_size - 1);
                                        -- The size of this table
                                        -- depends on the storage
                                        -- space available in the

-- init_num value used for echoing back the initialization parameters
read_init_formation : constant integer := 0;

-- Local variable declaration:
-------------------------------
-- The following variable is commented out. It appeared only in the
-- "high-level" used to read in the TOS table.  See below.
--  number_of_tos_table_octets: integer range 2 .. max_tos_table_size - 1;
octet_register:                 octet_type;
dump_init_information:          boolean;

-- The following procedure is used to communicate parameters for a
-- rendezvous with the entry Memory_out_request.
```

```
procedure Memory_out_request(
    request_type_formal:          memory_request_type;
                                -- Load_address or receive_datum_octet
                                -- or load_datum_octet.
    chunk_of_address_formal:      chunk_of_address_type;
                                -- Don't care when request_type_formal
                                -- receive_datum_octet.
    octet_formal:           in out octet_type)
                                -- Don't care when load_address.
  is
    request_message  :  memory_out_request_mess := new
                                       memory_out_request_rep;
    response_message :  memory_out_response_mess;
  begin
    request_message.request_type_formal      := request_type_formal;
    request_message.chunk_of_address_formal := chunk_of_address_formal;
    request_message.octet_formal             := octet_formal;
    Memory_Out_Request_Port_Def.
                        send(memory_out_request_port,request_message);
    Memory_Out_Response_Port_Def.
                    receive(memory_out_response_port,response_message);
    octet_formal := response_message.all;
  end Memory_out_request;

begin
  loop
    -- The following declare block replaces a Go accept statement
    declare
      go_request_message  : go_request_mess;
      go_response_message : go_response_mess;
      init_num_formal     : bit3;
      response            : out_response;
    begin
      -- The following 2 statements receive the in parameter to the
      -- Go accept statement
      Go_Request_Port_Def.receive(go_request_port,go_request_message);
      init_num_formal := go_request_message.all;
      response := sent_ok;                        -- Also means init_ok.
      if init_num_formal = read_init_formation
      then
        dump_init_information := True;
      else
        dump_init_information := False;
      end if;
```

```
-- Get from the server all of the addr_chunks needed to form the base
-- address in memory that holds the initialization parameters and
-- sends these chunks to the Memory module.
if not dump_init_information then
  for index in 1 .. init_num_formal
  loop
    -- The following declare block replaces a Srv_req accept
    -- statement
    declare
      srv_req_request_message   : srv_req_request_mess;
      srv_req_response_message : srv_req_response_mess;
      server_command_datum      : srv_command_type;
      response_to_server        : out_response;
    begin
      -- The following 2 statements receive the in parameter
      -- for the srv_req accept statement
      Srv_Req_Request_Port_Def.receive(srv_req_request_port,
                                   srv_req_request_message);
      server_command_datum := srv_req_request_message.all;
      Memory_out_request( -- Put chunk out to the Memory module.
          request_type_formal     => load_address,
          chunk_of_address_formal => server_command_datum,
          octet_formal            => dont_care_octet);
    end; -- end of Srv_req accept statement
  end loop;
end if;

-- Get the 6 individual initialization parameters (contained in the
-- next 8 octets received) from the Memory Module.
-- or, if init_num_formal is read_init_information,
-- send them back to the memory.
for index in 1 .. 8
loop
  if dump_init_information then
    case index is
      when 1 => octet_register := lnm_max_packet.lo;
      when 2 => octet_register := lnm_max_packet.hi;
      when 3 => octet_register := lnm_address_length;
      when 4 => octet_register := lnm_time_out.lo;
      when 5 => octet_register := lnm_time_out.hi;
      when 6 => octet_register := ack_type;
      when 7 => octet_register := local_net_type_of_service_table_row_size;
      when 8 => octet_register := number_of_local_net_types_of_service;
    end case;
  end if;
```

```
    if dump_init_information
    then
      Memory_out_request(
          request_type_formal      => receive_datum_octet,
          chunk_of_address_formal => dont_care_X_datum,
          octet_formal             => tos_table(index));
    else
      Memory_out_request(
          request_type_formal      => load_datum_octet,
          chunk_of_address_formal => dont_care_X_datum,
          octet_formal             => tos_table(index));
    end if;
    if not dump_init_information then
      case index is
        when 1 => lnm_max_packet.lo        := octet_register;
        when 2 => lnm_max_packet.hi        := octet_register;
        when 3 => lnm_address_length       := octet_register;
        when 4 => lnm_time_out.lo          := octet_register;
        when 5 => lnm_time_out.hi          := octet_register;
        when 6 => ack_type                 := octet_register;
        when 7 => local_net_type_of_service_table_row_size
                                   := octet_register;
        when 8 => number_of_local_net_types_of_service
                                   := octet_register;
      end case;
    end if;
  end loop;

  -- Read in type of service translation table (or write it out).

  declare
    row_number: integer range 0 ..
                         number_of_local_net_types_of_service;
    col_number: integer range 0 ..
                         local_net_type_of_service_table_row_size;
    index:      integer range 0 .. number_of_local_net_types_of_service
                         * local_net_type_of_service_table_row_size
                         := 0;
  begin
    row_number := 0;
    loop                   -- Outer loop reads all rows of TOS table.
      col_number := 0;
      loop                 -- Inner loop reads in one row of TOS table.
```

```
          if dump_init_information
          then
            Memory_out_request(
                request_type_formal      => receive_datum_octet,
                chunk_of_address_formal => dont_care_X_datum,
                octet_formal             => tos_table(index));
          else
            Memory_out_request(
                request_type_formal      => load_datum_octet,
                chunk_of_address_formal => dont_care_X_datum,
                octet_formal             => tos_table(index));
          end if;
          col_number := col_number + 1;
          exit when col_number =
                             local_net_type_of_service_table_row_size;
          index := index + 1;
          if  index > max_tos_table_size  then
            response := bad_srv_command;
            return;           -- Exit the current accept statement.
          end if;
        end loop;             -- End inner loop.

        row_number := row_number + 1;
        exit when row_number = number_of_local_net_types_of_service;
      end loop;               -- End outer loop.
    end;                      -- End declare block.
    go_response_message.all := response;
    Go_Response_Port_Def.send(go_response_port,go_response_message);
   end;                       -- End of init processing (Go accept).
  end loop;                   -- End of outer-most (inifinite) loop
 end Read_Init_Parameters;
 end RIP_Manager;
end RIP_Interface;
```

REFERENCES

[1] Brinch-Hansen, P.
The Nucleus of a Multiprogramming System.
Communications of the ACM 13(4):238-241, April, 1970.

[2] G. Kahn.
The semantics of a simple language for parallel programming.
In *Information Processing 74*, pages 471-475. IFIPS, North Holland, 1974.
The definitive reference on what we call Kahn processes'.

[3] Lindstrom, G., Organick, E.I., Klass, D., Maloney, M.
Ada Specifications for the DoD Internet Protocol: The INM OUT Submodule, Report No. 1.
Technical Report, Department of Computer Science, University of Utah, November, 1982.

[4] Organick, E.I.
Programmer's View of the Intel 432 System.
McGraw-Hill Book Company, New York, 1983.

[5] Organick, E.I., Keller, R. M., Lindstrom, G., Smith, K.F., Subrahmanyam, P.A., Carter, T., Klass, D., Maloney, M. P., Nelson, B.E., Purushothaman, S., Rajopadhye, S.
Transformation of Ada Programs into Silicon. Third SemiAnnual Technical Report.
Technical Report UTEC-83-026, University of Utah, April, 1983.