ABSTRACT

Creating a better environment for distributed programming is essential to speed up the development of distributed applications. The programming environment must address the following problems: naming of services in a distributed system, resource management, inter-process communication, and handling of exceptional conditions. This report describes a prototype system that solves those problems.

We view the distributed world as being composed of services and clients. Services have access to resources and clients acting on behalf of users obtain services. Our goals are to provide name and location independence and to provide high level interfaces. Our implementation strategy is to break the problem into three parts: (1) Naming Service, (2) PRIMOS Port and Process Manager, and (3) Local Agent.

The prototype system is installed on RES.C1, RES.C2, and RES.C4. The entire system runs as user code on top of PRIMOS REV. 13 and REV.19, requiring no modification of the existing system's kernels. A few working examples are used to illustrate the system.
# TABLE OF CONTENTS

1 Introduction ................................................. 3

2 An Informal Example. ........................................ 4
   2.1 Overview .............................................. 4
   2.2 The Server Process .................................... 4
   2.3 The Client Process .................................... 5

3 User Interfaces ............................................. 6
   3.1 Introduction .......................................... 6
   3.2 Minimal User Interface ............................... 7
      3.2.1 C$INIT .......................................... 8
      3.2.2 C$START ....................................... 9
      3.2.3 C$SEND ....................................... 10
      3.2.4 C$RECV ....................................... 11
      3.2.5 C$CLOSE ...................................... 12
      3.2.6 C$EXIT ....................................... 13
   3.3 Advanced User Interface .............................. 14
      3.3.1 Naming Service ................................. 15
         3.3.1.1 C$RGS .................................... 16
         3.3.1.2 C$CRS .................................... 17
      3.3.2 Error Handling ................................. 18
         3.3.2.1 C$ERR .................................... 19
         3.3.2.2 C$ERRST .................................. 20
         3.3.2.3 C$ERHN .................................. 21
         3.3.2.4 C$CHCN .................................. 22
      3.3.3 Flow Control and Multiplexing .................... 24
         3.3.3.1 C$LOCK .................................... 25
         3.3.3.2 C$UNLO .................................... 26

4 Additional Programming Tools .............................. 27
   4.1 Introduction to the Structure of Cache .............. 27
   4.2 Message Tracing. .................................... 27
   4.3 Error Logging ....................................... 28
   4.4 Debugging a Distributed Server ...................... 28

5 Conclusions ............................................... 29

Appendix: A PL/I Example .................................. 30
1. Introduction

Writing distributed programs for PRIMENET is difficult due to the low level interface of the X.25 Packet Layer and the lack of the naming service on the local network [PE-TI-729]. This report describes a prototype system that provides solutions to the following problems: naming of services in a distributed system, resource management, inter-process communication, and handling of exceptional conditions in the case of hardware or software failures. None of the ideas described in this report are particularly "original" or novel. We have talked about them for quite a while (PE-TI-729, PE-T-752, PE-TI-816). This project, however, is unique in combining all of these ideas in a relatively coherent manner.

The distributed operating system consists of three programs: NamingServer, PortManager, and LocalAgent. The NamingServer provides a mapping between the user oriented service names and the location of those services. For each node, the PortManager manages PRIMENET's ports and PRIMOS processes. For each user, the LocalAgent provides a convenient and easy to use interface to the distributed system.

This report has three major sections: Section 2 introduces the minimal set of primitives by an example of a network server. The complete PL/1 program is contained in the Appendix. Section 3 is the reference manual to the user interface. Section 4 describes additional programming tools to aid the debugging and performance analysis of distributed programs.

The system installation is trivial: each node must have the PortManager running as a phantom; a user program must load the PL/I library ConLib.
2. An Informal Description of the Distributed Program

2.1. Overview

The purpose of this section is to introduce the minimal set of programming primitives for distributed applications. The example used throughout this section is the distributed version of the "Adventure" program. (The Appendix contains PL/I sources of the program). The distributed version is implemented by splitting the centralized version of the program into two parts: client and server. The distributed version of the program is very robust: each part of the program gracefully recovers from failures in the other part.

The server is essentially the centralized version of the program except it has the new network service interface that replaces the old terminal I/O interface. Instead of communicating with a terminal, the server communicates with the client. The client is a virtual terminal: upon the server's input request it prompts the user for more input from the keyboard; upon the server's output request it displays data on the screen.

The distributed program is very robust: failure of the server causes the client to search for another machine to create the same service; failure of the agent is ignored by the server, which simply waits for a new agent to come up. This flexibility is easily implemented due to the nature of the application: the occasional loss of output data may be tolerated by a user who will repeat the input request; the lost service will be automatically replaced by the service on another machine. The resulting program survives crashes of machines with a very high probability (the crash of two machines simultaneously will disable the running program).

2.2 The Server Program

The server communicates with the client process, which acts like a virtual terminal. To display data on the user terminal, the server program calls procedure C$SEND that sends data to the client which in turn will eventually output data to the terminal. To receive user input data, the server first calls procedure C$SEND that sends a request to the client for more input and then waits for the data to arrive with procedure C$RECV.

The server program easily recovers from various failures on the part of the agent or on the part of the link connecting the server to the agent. A failure in procedure the C$SEND is ignored by the server, which uses the procedure C$ERST to provide an "umbrella" for network errors. A failure in procedure C$RECV at one of the connections remains unnoticed because the server waits for a message to arrive at any connection including any new connection yet to be opened.

The crash of an agent during the moment when the server calls C$SEND may cause some output data to be lost. This is an acceptable situation
since the user will not see the normal response from the system and will retype the request. The crash of an agent during the moment when the server calls C$RECV is invisible to the server because no information is lost.

2.3 The Client Program

To create a service on a remote machine the client calls procedure C$CRSR that returns an identifier of the connection between the client and the server. (The service must have been registered before with the procedure C$RGSR). Procedures C$SEND and C$RECV use the identifier to send and receive data.

To protect the client from various error signals that may occur at the connection the client uses function C$ERST. Any error will be trapped to the main program where the service is again established on another machine.
3 User Interfaces

3.1 Introduction

This section is the reference manual for all the procedures in the utility package. The PL/1 header "NetUser.Pl1" contains the entry definitions of these procedures.

This section contains two major subsections: Subsection 3.2 describes the minimal set of functions (six PL/I procedures) necessary to write distributed programs. A more advanced user may read the Subsection 3.3 that describes various miscellaneous routines.
3.2 Minimal User Interface

This section describes the six functions that are necessary for two processes to communicate. The functions are based on the concept of a connection that exists between two communicating processes.

Any process must call procedure C$INIT to initialize the network environment and procedure C$EXIT to exit cleanly from the network environment. Then, the client calls procedure C$STRT, which starts and establishes the connection between the client and the server. The server simply waits for a message to arrive at any connection. Once the connection is established, both process may send and receive messages in any direction. A process sends a message with the function C$SEND; a process receives a message with the function C$RECV. Both processes release resources associated with the connection by calling the function C$CLOS.

All utilities use a connection identifier as a parameter, thereby, uniquely identifying one open connection. As an exception, the value "minus one" (-1) represents any open connection. For example, a process passing the value "minus one" to the function C$RECV waits for a message to arrive at any open connection. Once the message arrives at a particular connection, function C$RECV returns the data as well as the identifier of the connection.

The description of each function has a PL/I entry statement. (Appendix has listings of header 'NetUser.PL1', which contains PL/I entries of all user functions). The variables in the parameter description conform to the following convetions: a scalar variable is a short integer; a buffer is a fixed length array; the length of the buffer always represents the number of bytes. Some functions have parameters with a specified value "par =value". Different parameter values will cause different actions to be taken by the function.
The function "Initialize" must be called by any process at initialization time. At execution time the function parses the PRIMOS Command Line that contains system data. The PortManager uses the command line to communicate with a distributed server (see Section 4 about the exact structure of the command line).
c$exit

declare c$exit entry;

The function "exit" must be called by any process that wishes to cleanly exit from the network environment. Before completing a user program, the process notifies the PortManager.
PRIMELINK, a High Level Service on Top of Today's PRIMENET

\texttt{c\$strt}
\begin{verbatim}
declare c\$strt entry(char(MAXSTR), fixed bin, char(MAXSTR), fixed bin, fixed bin) returns(fixed bin);
connection = c\$strt(target\_node, len1, file\_name, len2, type);
\end{verbatim}

The function "Start remote process" is performed in two steps: the first step creates the remote process according to the specified code segment or command file (the code segment is assumed to be a type SAM segment directory). The second step creates a connection between the caller of C\$STRT and the remote process. The function returns the connection identifier that is used during the transfer of data. The function is based on one PRIMOS routine, PHANT$, for creating a new process and on three PRIMENET procedures, X\$CONN, X\$GCON and X\$ACPT, for creating a connection.

connection = an identifier of the newly created connection.

target\_node = a buffer containing the ASCII name of the PRIMENET system that is the target of this connection request.

len1 = the length of target\_node.

seg\_name = a buffer containing the ASCII name of the code segment or command file that is started as a process.

len2 = the length of seg\_name.

type = (=1), each time start the service for private use;

\begin{itemize}
  \item (=2), start the service if it does not exist,
  \item otherwise connect to the existing service;
\end{itemize}

(default: 1).
PRIMELINK, a High Level Service on Top of Today's PRIMENET

The procedure "Send and wait" sends a message at the given connection. If the message is queued successfully, the sender returns immediately; otherwise, the sender is suspended. The procedure is used by dedicated processes that are willing to suspend their computations until the system can queue their messages. It is based on two PRIMENET procedures: X$TRAN for sending data and X$WAIT for waiting until the message is successfully queued.

c$send

declare c$send entry(fixed bin, char(MAXBUF), fixed bin);
call c$send(connection, data, len);

connection = an identifier of the connection used
to send the buffer of data.
data = the data buffer.
len = the length of data.
PRIMELINK, a High Level Service on Top of Today's PRIMENET

\texttt{c$recv}

\texttt{declare c$recv entry(fixed bin, char(MAXBUF), fixed bin, fixed bin)}
\texttt{return(fixed bin);} 
\texttt{res\_con = c$recv(connection, data, len, timeout)}

The function "Receive and copy into the user's space" attempts to receive a message at the given connection and place the data into the user provided buffer. If no message arrives in a specified period of time, the function times out and returns to the user. The function is based on two PRIMENET procedures: \texttt{X$RCV} for receiving a message and \texttt{X$WAIT} for waiting until the message arrives.

\texttt{res\_con = output;} an identifier of the connection on which the message has been successfully retrieved;

\texttt{connection = input;} an identifier of the connection on which to wait for a message.

\texttt{data = input;} a pointer to the user provided buffer where the received data is placed.

\texttt{len = output;} the actual length of the received buffer.

\texttt{timeout = input;} the maximum time interval in seconds that the process is waiting for the message to arrive.

\texttt{timeout = input;} the maximum time interval in seconds that the process is waiting for the message to arrive.

\texttt{timeout = input;} the maximum time interval in seconds that the process is waiting for the message to arrive.

\texttt{(default: -1).}
c$clos

declare c$clos entry(fixed bin);

call c$clos (connection)

The procedure releases all system resources associated with the connection. The procedure is based on PRIMENET subroutine X$CLR.

connection = an identifier of the connection.
3.3 Advanced User Interface

This Section contains three independent subsections: Subsection 3.3.1 describes functions for registering and starting new services; Subsection 3.3.2 describes functions for an advanced error handling; Subsection 3.3.3 has functions for multiplexing and flow control of a stream of messages.
3.3.1 Naming Service

All services in a distributed system may be described by a unique logical name that is independent of location, thus preventing user sensitivity to service location. PE-TI-729 has a more detailed discussion of naming architecture. This section describes two functions, C$RGSR and C$CRSR, that support such an architecture. Function C$RGSR allows a user to register a service on a given machine; function C$CRSR uses that name in order to connect to that service.
The function "Register a new Service" installs a new service in the distributed system. The service may later be used with function C$CRSR, thereby, providing for Name-Location independence.

name = a buffer containing the ASCII name of the service.
len1 = the length of name.
target_node = a buffer containing the ASCII name of the PRIMENET system where to register the service.
len2 = the length of target_node.
seg_name = a buffer containing the ASCII name of the command file or code segment of the new service.
len3 = the length of seg_name.
password = a buffer containing the ASCII name of the password of the service.
len4 = the length of password.
c$crsr

declare c$crsr entry(char(MAXSTR), fixed bin,
                     char(MAXSTR), fixed bin);

call c$crsr(name, len1, password, len2);

The function "Create a Service" locates and creates a service that has been previously registered.

name = a buffer containing the ASCII name of the service.
len1 = the length of name.
password = a buffer containing the ASCII password of the service.
len2 = the length of password.
3.2.3 Error Handling

Within a single process, procedure C$ERR signals the occurrence of an error condition with a given severity code. Procedure C$ERR typically signals a network error condition with a certain severity code. Any user defined error might also be signaled with the same procedure C$ERR. To protect a procedure call from error signals, a process may call function C$ERST to provide an "umbrella" for error conditions signaled with procedure C$ERR. A more powerful procedure C$ERHN is similar in function to procedure C$ERST, but it provides an error handler to be called to "fix" the error. If the error condition is indeed "fixed", the user program continues from the point of interruption.

Another way to handle network errors gracefully is to wait for a message to arrive at any connection. A change in status of one connection does not affect the status of the running process. The process simply waits for the message to arrive at another connection. To enable the process to cleanup its state, function C$CHCK checks the status of the connection.
c$err

declare c$err entry(fixed bin, fixed bin, char(MAXSTR), fixed bin) options(variable);
call c$err(severity, error_code, data,len);

Procedure "Error" signals the occurrence of an error condition. The error condition is propagated upwards in the stack. At a higher level, function C$ERST might "catch" the error condition by returning control to the caller of C$ERST. The "catch" occurs if the function C$ERST is called with a higher severity code than the procedure C$ERR is called. For example, the following call to procedure C$ERR,

CALL C$ERR(50, ...)

is caught by the following call to function C$ERST,

res = C$ERST(100, ...)

If no call to function C$ERST is present, the error condition propagates to the top level and aborts the program. If the error is caught by function C$ERHN, it invokes the specified exception handler with the user provided data. The exception handler may "fix" the error and return to the point of interruption or fail to "fix" and allow the error condition to propagate to a higher level at the stack. The option "options(variable)" allows a user to specify only the first two arguments: severity and error_code.

severity = severity code as chosen by the user;
(=0), do nothing but log the event.

error_code = the user generated error code.
data = a user buffer of data that is passed to the exception handler in C$ERHN (optional).
len = the length of data (optional).
PRIMELINK, a High Level Service on Top of Today's PRIMENET

C$ERST

Declare C$ERST entry(fixed bin, fixed bin, fixed bin,
entry(fixed bin, char(MAXSTR), fixed bin),
fixed bin, char(MAXSTR), fixed bin)
returns(fixed bin) options(variable);

Err = C$ERST(severity, error_code, res,
user_function, con, data, len);

The function "Error Set" protects user_function from error signals
with a lower severity (see procedure C$ERR). The option
"options(variable)" allows a user to pass a procedure with any
number of arguments of any type.

Err = (=0), no error was caught;
otherwise, an error was signaled,
see error_code. severity = a user supplied severity code.

Error_code = error code that was returned by C$ERR.
Res = returned result of user_function.
User_function = external procedure to be protected.
Con = an identifier of the connection.
Data = the buffer of data.
Len = the length of data.

N.B. These three arguments represent a typical
example of user_function.
The function "Error Set and provide an error Handler" differs from function C$ERST only in that it provides an exception handler that may "fix" the error.

The handler is invoked upon occurrence of the error and returns a boolean value indicating whether or not to abort the procedure call in progress. If the returned value is TRUE, the process continues from the point of interruption; otherwise, the process returns to the caller of user_function, a procedure at the higher level in the stack.
PRIMELINK, a High Level Service on Top of Today's PRIMENET

$\text{c$chcn}$

Declare $\text{c$chcn}$ entry(fixed bin, (2)fixed bin)

$\text{returns(fixed bin)}$;

$\text{res} = \text{c$chcn$(connection, status)}$;

The function "Check Connection" checks the status of the connection.

$\text{res} =$ returned result of the function;

$(\approx 0)$, the connection is dead;

$(\approx 0)$, the connection is alive.

$\text{con} =$ identifier of the connection.

$\text{status} =$ the returned connection status array that is used to hold virtual circuit status

(see PRIMENET manual).
3.3.4 Flow Control and Multiplexing

PRIMENET can hold only a limited number of messages in transit. If a sender exceeds the system's capacity to hold messages in transit, the sender is suspended (see procedure C$SEND). A receiver may force a particular sender to be suspended by locking the queue at a given connection. This is convenient for a receiver waiting for a message to arrive at any open connection. A call to procedure C$LOCK locks the queue at a given connection; a call to procedure C$UNLOCK unlocks the connection.
c$lock
declare c$lock entry(fixed bin);
call c$lock(connection);

The procedure "Lock" locks the connection with the effect that no messages on the specified connection are accepted.

connection = identifier of the connection.
c$unlo

declare c$unlo entry(fixed bin);

call c$unlo(connection);

The procedure "Unlock" unlocks the connection allowing messages on
the specified connection to be accepted.

connection = identifier of the connection.
4 Additional Programming Tools

4.1 Introduction to the Structure of Cache Files

The Distributed Operating System is controlled by three tables: (1) Servers.Cache which is accessed by LocalAgent; (2) Servers.Node which is accessed by PortManager; (3) Servers.All which is accessed by NamingServer. Each table is encoded as an ASCII files that may be edited by an authorized user or modified by an associated program. The existence of each file is optional, although the system may create it during execution of a program. For example, Servers.Cache is updated by LocalAgent that remembers the most frequently used services. The absence of this file may result in additional requests being sent to the NamingServer. The file Servers.Node contains all the services registered on this particular node. The file is updated by the PortManager which adds new services to the system. The file is created by users who register new services.

The service tables are encoded using conventions similar to those by PRIMOS for encoding command lines. Each token is encoded by a key word and value pair. The following example is a description of one server in either the file Servers.Cache or Servers.Node.

-N Server1 -L location -S SegmentFile
-M message_trace_option
-E error_logging_option -A access_control_list
-P port_number -N Server2

To change the characteristics of a server, the user changes the file Servers.Node; to change the Agent's viewpoint of the server, the user changes Servers.Cache file (see Section 4.4 on debugging communicating processes). Finally, to define characteristics of an agent the user enters parameters in a command line. For example, to invoke an agent with the message trace option the user enters the following command:

SEG AGENT -M 1
4.2 Message Tracing

-M 1 this option enables the time-stamped message trace. facility. For each message the system stores the following PL/I record:

```pli
declare
  1 Trace,
  2 UserNumber fixed bin,
  2 Connection fixed bin,
  2 Type fixed bin,
  2 MessageCount fixed bin,
  2 Len fixed bin,
  2) Time fixed bin(31),
  2) Blank fixed bin;
```

The system collects the message header in a PRIMOS segment pointed to by the variable Trace_Seg. (Include entry "sy Trace_Seg SEGMENT_NUMBER) in the load command). After completing the program's execution, the trace information may be dumped to a file.
4.3 Error Logging

-e 1 opens an error logging file for that process.

All calls to the procedure C$ERR are registered within
the file "Error.log".

4.4 Debugging the Interaction Between

a Pair of Communicating Processes

-p port-number, a debugging option allowing a user to
assign manually a port to a service.

To invoke the server, for example, a user types

SEG SEGMENT_FILE -p 5

to invoke the agent, the user must edit the file Servers.Cache so as to
modify the entry corresponding to the server that uses the chosen port
number.
6. Conclusions

The purpose of this project was both to develop a minimal prototype system and to develop a methodology for distributed programming at PRIME. This document describes all the primitives of the system and also gives a few examples of distributed programs. The sources of some such programs appear in the Appendix.

Our experience with a real system and real examples indicates that our methods are sound and that even crude exception handling mechanisms provides adequate means for writing robust programs. The next step is to build more applications and gain more experience with distributed programs.
This program is a client for the distributed "Adventure" server. It acts like a remote virtual terminal. Upon server requests, it prompts the user for more input; otherwise, it displays the data on the screen. The procedure HandleMessage is protected from network errors with the function C$ERST.

The client communicates with two servers: "Adv1" and "Adv2". If one server dies, the client restarts another server. Both servers must have been previously registered.

The main program waits for a request in an infinite loop. The request is handled by procedure HandleMessage. All network errors are protected with function C$ERST. If an error occurs, the program an prints an error code, and the message "connecting to another system". Thereafter, it creates the alternate server.
PRIMELINK, a High Level Service on Top of Today's PRIMENET PE-TI-900

/* CLIENT PROCESS */

main: procedure;

declare
 (code, res, len, len4) fixed bin,
  outcon fixed bin,
  current fixed bin,
  tempstr char(MAXSTR),
  Games(2) char(MAXSTR),
  Msg char(MAXBUF);

/* MAIN PROGRAM */

put skip list('started');

call c$init;
Games(1) = 'Adv1';
Games(2) = 'Adv2';
len = 4;
put skip list('initialized');

current = 1;
tempstr = Games(current);
outcon = c$crsr(Games(current), len, '', 0);

put skip list('connected');

call EncodeFortranString(Msg, 'yes', 3);
call c$send(outcon, Msg, 3*4); /* the length of FORTRAN buff */

*/
do while('1'B);
  if c$erst(SYSERR, code, res, HandleMsg, outcon, Msg, len) ^= 0 then
    do;
      put skip list('error:=', code, ' connecting to another sy stem');
      if current = 1
        then current = 2;
      else current = 1;
tempstr = Games(current);
  outcon = c$crsr(Games(current), len, '', 0);
  put skip list('connected');
  put skip list(' ');
end;
end;
This procedure implements the virtual terminal. Upon server requests, (TIOFIX(1) = -1), the procedure prompts the user for more input; otherwise, the procedure displays data on the screen. In both cases, the data must be converted from FORTRAN string representation into PL/I string representation.

HandleMsg: procedure(con, TIOArr, len) returns(fixed bin);
declare
  TIOArr char(MAXBUF),
  con,
  len fixed bin;
declare
  TIOFix(HALFBUF) fixed bin defined(TIOArr),
  timeout fixed bin,
  i fixed bin,
  FirstNum fixed bin,
  Str char(128),
  VarStr char(128) var;

  timeout = c$recv(con, TIOArr, len, 500);
  if timeout = 0 then call c$err(SYSERR, NORESPONSE);
  len4 = len/4;
  FirstNum = TIOFix(1);
  if FirstNum ^= -1 then
    do;
      call DecodeFortranString(TIOArr, Str, len4);
      put skip edit(substr(Str, 1, len4)) (a);
    end;
  else
    do;
      get list(VarStr);
      len4 = length(VarStr);
      call EncodeFortranString(TIOArr, VarStr, len4);
      call c$send(con, TIOArr, len4*4); /* the length of FOR
TRAN buff */
    end;
  return(1);
end HandleMsg;

EncodeFortranString: procedure(TIOArr, VarStr, len4);
declare
  TIOArr char(MAXBUF),
  1 TIOMsg defined(TIOArr),
  2 REC(128),
  3 Data char(1),
  3 BlankData char(3),
  VarStr char(128) var,
  len4 fixed bin;
declare
i fixed bin;
/* encode into FORTRAN LONG INTEGER FORMAT */
do i=1 to len4;
  TIOMsg.Data(i) = substr(VarStr, i, 1);
  TIOMsg.BlankData(i) = ' ';
end;
len4 = len4 + 1; /* append a blank */
TIOMsg.Data(len4) = ' ';
end EncodeFortranString;

DecodeFortranString: procedure(TIOArr, Str, len4);
declare
  TIOArr char(MAXBUF),
  1 TIOMsg defined(TIOArr),
  2 REC(128),
  3 Data char(1),
  3 BlankData char(3),
  Str char(128),
  len4 fixed bin;
declare
  i fixed bin;
/* decode from FORTRAN LONG INTEGER FORMAT */
do i=1 to len4;
  substr(Str, i, 1) = TIOMsg.Data(i);
end;
end DecodeFortranString;
/* ADVENTURE-SERVER PROCESS, NETWORK INTERFACE */

This set of network interface routines replaces the original terminal I/O routines. Incoming data is received with procedure M$RCCP; output data is sent with procedure M$SEND. The variable NetOn indicates whether the network package is being used.

M$RCCP: procedure(con, buf, len);

declare (con, len) fixed bin(31);
declare buf char(512),
    NetOn fixed bin(15) external,
    len1 fixed bin(15);
if NetOn ^= 0 then
    con = c$recv(-1, buf, len1, -1);
    len = len1;
return;
end M$RCCP;

M$SNWT: procedure(con, buf, len);

declare (con, len) fixed bin(31);
declare buf char(512),
    dum fixed bin,
    (code, res) fixed bin,
    NetOn fixed bin(15) external;

if NetOn ^= 0 then
    dum = c$erst(SYSERR, code, res, c$send, con, buf, len);
return;
end M$SNWT;

M$INIT: procedure(con);

declare con fixed bin(31);
declare buf char(512),
    NetOn fixed bin(15) external init(1),
    len fixed bin(15);

if NetOn ^= 0 then
    do;
        call c$init();
        con = c$recv(-1, buf, len, -1);
    end;
end M$INIT;
PRIMELINK, a High Level Service on Top of Today's PRIMENET PE-TI-900

/* NETUSER.PL1 HEADER */
/* This is a standard header typically used by a server */

MAXSTR by 60,
MAXBUF by 512,
HALFBUF by 256,

/* common errors */
FATAL by 100,
SYSERR by 50,
USRERR by 25,
NOTIMP by -1,
BADPROT by -2,
NORESPONSE by -3;

/* msg(1) values: request and reply */
/* reply */
REP_OK by 0,
REP_ERR by 1;

/* request */
declare
(addr,null, substr, index, length) builtin;

declare
c$err entry(fixed bin, fixed bin),
c$send entry(fixed bin, char(MAXBUF), fixed bin),
c$recv entry(fixed bin, char(MAXBUF), fixed bin, fixed bin) returns(fixed bin),
c$clos entry(fixed bin),
c$erst entry(fixed bin, fixed bin, fixed bin, entry(fixed bin, char(MAXBUF),fixed bin) return
s(fixed bin),
   fixed bin, char(MAXBUF), fixed bin) returns(fixed bin),
   c$erhn entry(fixed bin, fixed bin, fixed bin,
         entry(fixed bin, char(MAXBUF), fixed bin) return
s(fixed bin),
   fixed bin, char(MAXBUF), fixed bin,
   entry(char(MAXSTR), fixed bin) returns(fixed bin) returns(fixed bin))
c$chn entry(fixed bin, (2)fixed bin) returns(fixed bin);

...
fixed bin, fixed bin)
    returns(fixed bin),
    c$rgsr entry(char(MAXSTR), fixed bin, char(MAXSTR), fixed bin,
    char(MAXSTR), fixed bin, fixed bin),
    c$crsr entry(char(MAXSTR), fixed bin, char(MAXSTR), fixed bin,
    returns(fixed bin);
    declare
    DefaultArg entry(fixed bin, fixed bin, fixed bin);
    declare
    Debug fixed bin external;