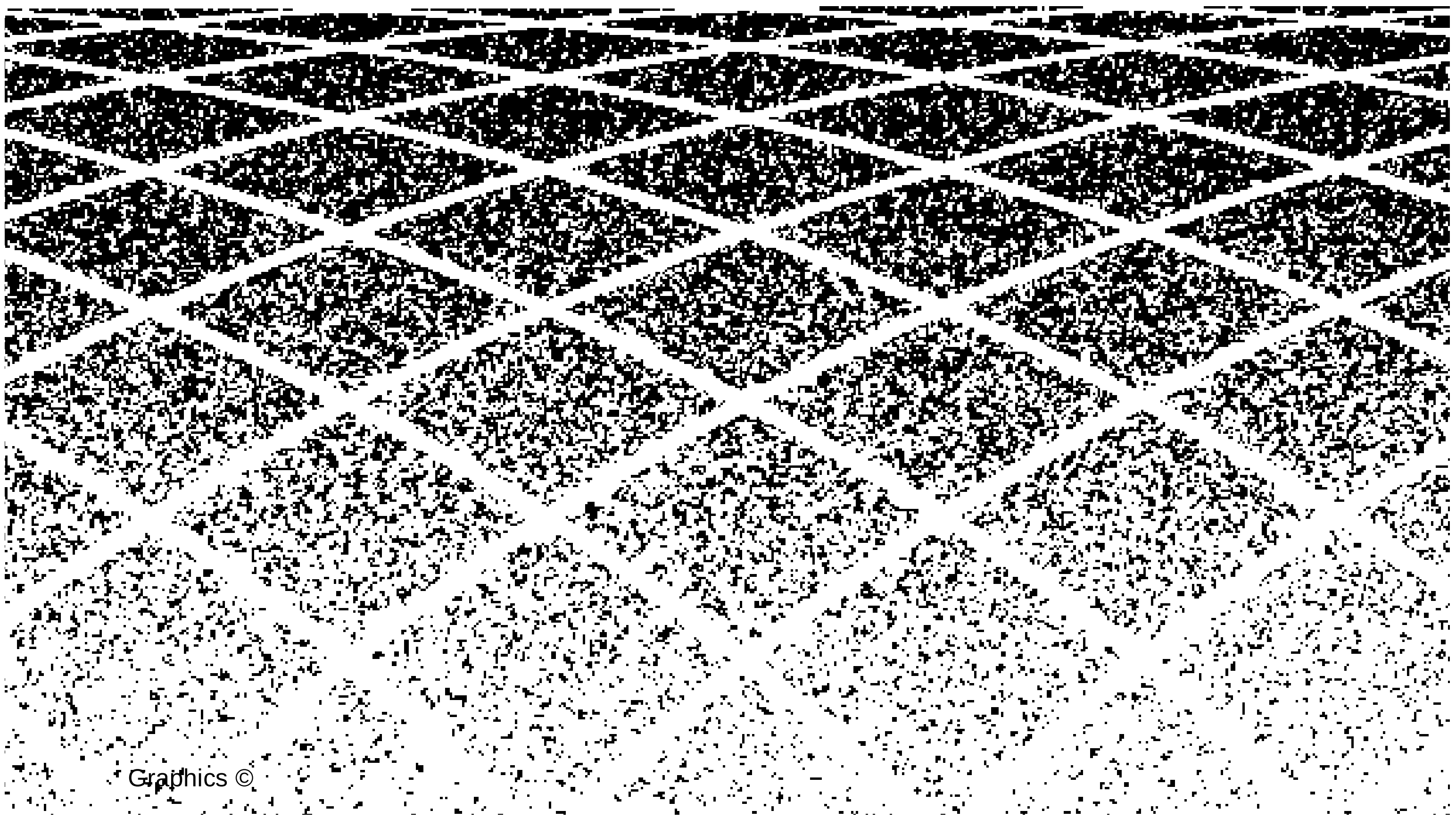
585-350-215
Issue 1
May 1994

Asynchronous Host Toolkit for CONVERSANT® VIS Version 4.0



Blank Page

Contents

Contents

Asynchronous Host Toolkit for CONVERSANT® VIS Version 4.0

1. Document Overview

This document describes the Asynchronous Host Toolkit feature developed for the CONVERSANT Voice Information System (VIS) Version 4.0 software. Section 2 provides an overview of the asynchronous Data Interface Process (DIP). Section 3 gives insight into the design and structure of the processes associated with the asynchronous DIP. Installation details are outlined in section 4. Section 5 presents a sample Script Builder application and simulated host program that can be used to validate the functionality of the DIP or as a starting point for customized development. Section 6 covers the steps necessary to modify, recompile and install customized versions of the asynchronous DIP. A listing of all the source code, header and make files associated with the asynchronous DIP can be found in the Appendices that follow section 6.

1.1 Toolkit Intent

This toolkit is intended to provide a C/UNIX/CONVERSANT familiar programmer with all the pieces necessary to develop an efficient asynchronous interface to their host computer. It does require development on the host computer to match the protocol outlined in this document.

This package was developed by the AT&T Custom Services Group (CSG) and has been successfully integrated into numerous customer applications. The CSG provides customized CONVERSANT VIS solutions on a bid basis. If you desire further information on how the CSG could assist you in this or other CONVERSANT VIS projects, contact your local AT&T Account Executive.

2. Asynchronous DIP Overview

The Asynchronous Host Toolkit provides a mechanism for CONVERSANT VIS scripts to communicate with customer host systems which support asynchronous communication over RS-232 serial ports.

The Asynchronous Host Toolkit is implemented using a client/server mechanism whereby the client process resides on the CONVERSANT VIS and the server process resides on the customer's host system.

2.1 Communications Process Overview

The following diagram depicts a high level view of the interface architecture:

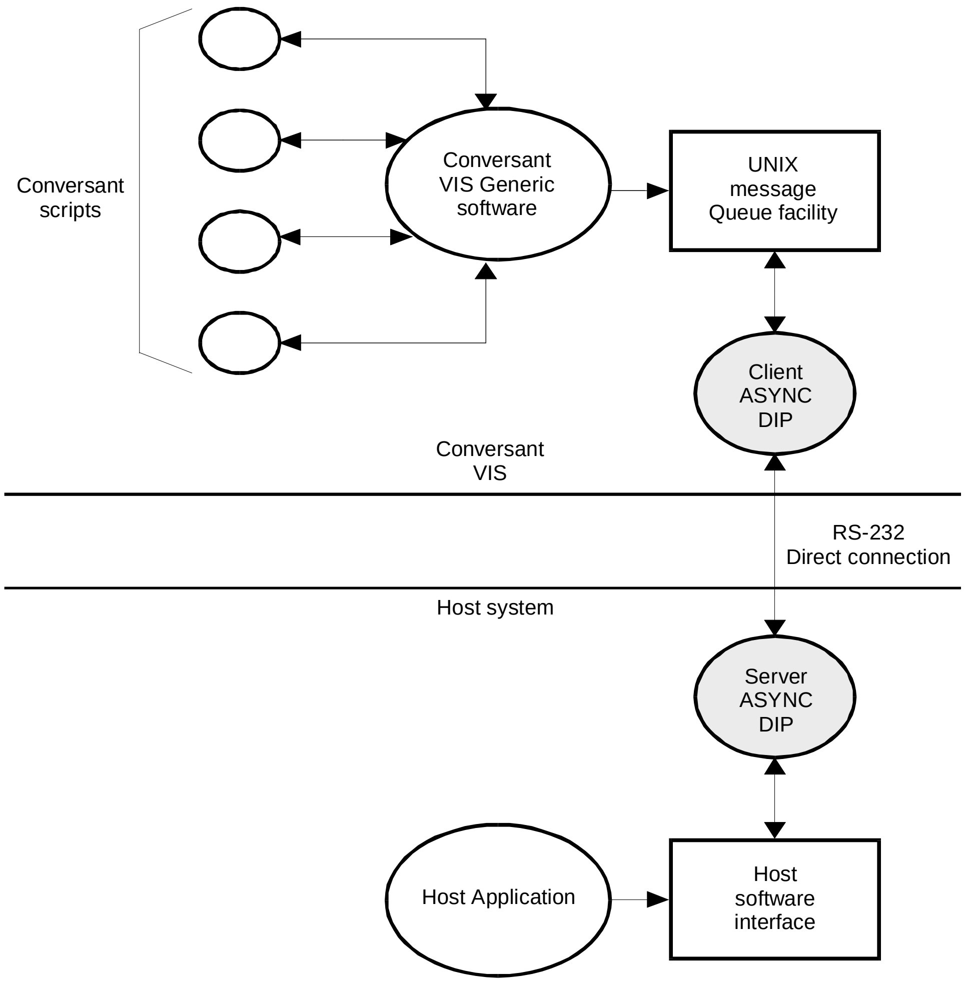


Figure 1.

A general description of the communications process follows:

- A CONVERSANT VIS voice script which requires data from the host executes an external function which causes the CONVERSANT VIS generic software to place a data request on a UNIX message queue read by the client asynchronous Data Interface Process (DIP).

- The client asynchronous DIP formats a data request message, and transmits the message over the RS-232 link.
- The server asynchronous DIP reads the data request message, sends a positive acknowledgment to the client, processes the data request through its interface to the host application, receives the data from the host application, formats a data response message, and transmits the response message over the RS-232 link.
- The client asynchronous DIP reads the data response message, sends a positive acknowledgment to the server, and sends the response data to the voice script via a UNIX message queue read by the CONVERSANT VIS generic software.

There is a special function code, 002, provided for host initiated requests.

2.2 Implementation Details

The Asynchronous DIP is implemented using two processes: a send process and a receive process. Furthermore, the DIP can be configured to operate on a single line or on two lines. The communications processes are hsend1 and hrec1 for line 1. These use external function hostreq1 for communication with the application script. For line 2, the communications processes are hsend2 and hrec2, using external function hostreq2 for communication with the application script.

Communication with the voice application and between the DIP processes is performed via CONVERSANT VIS DIP numbers which map into UNIX message queues. The Asynchronous Host Toolkit utilizes the Dynamic DIP feature.

Script Builder applications communicate with the Asynchronous Host Toolkit via the external functions 'hostreq1' and/or 'hostreq2', which use the following format:

Input:

ARG1	header information; a character string of the format MMSSFFF where MMSS is the date stamp (minutes and seconds) and FFF is the function code (application specific)
ARG2	the input character string with fields separated by an arbitrary delimiter or fixed length fields - this format is completely open to the developer
ARG3	alarm interval (time-out value >= 10 seconds) provided in a numeric field. The DIP uses an ACK/NAK time-out of about 1/4 this value.

Output:

ARG4	the output character string with fields separated by an arbitrary delimiter

Return Value:

the response code converted to a numeric

The response code is set by the host except for four cases:

'-3'	too small a time-out (ARG3) value passed to the external function from the application.
'-2'	the external function timed out without a response back from the DIP within ARG3 seconds. Host acknowledged request but didn't respond fast enough.
'-1'	the DIP send process (hsend1 or hsend2) was not running.
'999'	the DIP was unable to send the request to the host (disconnected from the host or the host not acknowledging the request).
'998'	the DIP received two NAKs from the host (data transmission facilities are suspect).

During initial installation, or via the command "asyncDIP_cf", the pair of asynchronous data interface processes can be configured to operate in one of four possible modes:

1 -	turn on only the DIP for the hostreq1 host line.
2 -	turn on only the DIP for the hostreq2 host line.
S -	turn on both the hostreq1 and hostreq2 line DIPs, but have them operate independently.
D -	Have the hostreq1 and hostreq2 line DIPs operate together, each prepared to backup the other in case one can't communicate on its host line.

In dependent ('D') mode of operation, the following DIP activities take place:

- The application sends the request to the external function, hostreq1 or hostreq2.
- The external function sets its time-out value to ARG3 seconds and sends another time-out value of about 1/4 ARG3 seconds to the DIP send function (hsend1 or hsend2).

- The DIP sets its time-out value and sends the request to the host over the appropriate line, then waits for acknowledgment.
- The host sends its ACK or NAK to the DIP receive process (hrec1 or hrec2), which promptly forwards it to its send counterpart.
- If a NAK, the send process re-sends the request to the host; if an ACK, it does nothing more.
- If the DIP primary send process succeeds in getting acknowledgment within 2 tries, the host response will be delivered to the external function via the DIP primary receive process.
- If the primary send process fails twice in getting acknowledgment from the host, it will send the request to its alternate DIP.
- If a DIP send process gets a request from its alternate (not TSM), it turns on a bit in the channel field of the header, causing the digit in the units position to become a lower case letter 'p' through 'y'.
- When a DIP receive process gets a response with that bit turned on, it resets the bit to off (and hence restores the channel field value), and sends the response back through TSM to the alternate external function.
- If the primary send process fails to communicate to either its host or to its alternate DIP, it will send a '999' return value back to its primary external function.
- If the send process is acting as alternate for a request, and it fails twice to get acknowledgment from its host, it will send a '999' return value back to the alternate external function.
- If the external function gets a response whose header time stamp doesn't match the time stamp of the request, it will set its time-out value to 1/2 of the original value and send a RETRY request to the DIP send process.
- If a DIP send process gets a RETRY request, it will throw it away, allowing the external function to wait on the original request.

In 2-line separate ('S') mode of operation, all activities take place as explained above, except:

- A primary send process will NOT forward a request to its alternate.
- The header channel field will always show numeric values.
- A receive process will never get a response for an alternate DIP.
- The effective overall DIP ACK/NAK time-out will not be more than 1/2 of the time-out set by the application.

2.3 Asynchronous Port Communications Attributes

The async DIP sets the following port attributes:

```
iflag = ( IGNPAR | IGNBRK | IXON | IXOFF | ISTRIP )
      oflag = 0
      cflag = ( CREAD | B9600 | CS8 | CLOCAL )
      lflag &=~ICANON
      lflag &= ~ECHO
      cc[4] = 0 for writing, 1 for reading
      cc[5] = 30 for writing, 90 for reading
```

A detailed explanation of these parameters is provided in the UNIX System V System Administration Reference manual page TERMIO(7): termio - general terminal interface.

The parameters are summarized below:

iflag: *Input control parameters*
IGNPAR -Ignore characters with parity errors
IGNBRK-Ignore break condition
IXON-Enable start/stop output control
IXOFF-Enable start/stop input control
ISTRIP-Valid input characters are stripped to 7 bits

cflag: *Hardware control*
CREAD -Enable receiver
B9600 -9600 baud
CS8 -8 bits
CLOCAL -Direct connection - no modem control

lflag: *Terminal function control*
~ICANON -Disable canonical processing
~ECHO -Disable echo

cc[]: *Control characters*
cc[4] -Minimum number of characters to be transmitted at a time
cc[5] -Maximum time between input character transmissions

3. Design Considerations

The Asynchronous DIP was designed in two primary pieces, a reader and a writer. This implementation allows one process (the writer) to sleep on the message queue waiting for requests from the voice script while the other (the reader) sleeps on the TTY port waiting for responses from the host. This split allows much greater throughput since the writer does not have to wait for the host response before servicing another request from the voice scripts. This section describes in detail how the processes work together to handle multiple, simultaneous requests for data.

3.1 The Writer Process (HSEND)

The writer process, HSEND, is responsible for receiving data from the voice scripts, formating it, and then writing it out to the TTY port. The source code for HSEND can be found in Appendix A.

3.1.1 Start Up

Upon initialization, HSEND first verifies the numbers of parameters passed into it. This should always be correct if the tool asyncdip_CF tool is used to modify the configuration. Next, the process checks to see how it was invoked; either as hsend1 or hsend2. It then gets a UNIX message queue assigned to it accordingly. Note that the names and location of the executables (/vs/bin) is critical to the functionality of the DIP.

After the successful acquisition of a message queue and the initialization of log messages, HSEND attempts to connect to and condition the TTY port specified as an argument. Failure to accomplish these tasks will result in the DIP terminating and generating corresponding Logger messages. Note that since the DIP conditions the TTY lines itself, the port should be set to device type NONE through FACE.

The final step in the start up phase is to determine if the DIP has a backup DIP running in the Duplex mode. If so, this DIP gets the message queue key of the other backup process.

3.1.2 Transaction Loop

Once the start up activities have successfully completed, HSEND goes into an infinite loop sleeping/reading its message queue. Messages to this queue can come from three different sources; TSM, HREC, and the backup writer process. The types of messages and how they're handled are outlined below.

3.1.2.1 TSM Messages

TSM is a CONVERSANT VIS process that executes the Script Builder applications. The application through the use of a dbase call requests that TSM send a message to a particular message queue. Details associated with that interface are described in the "3.3 The External Function (HOSTREQ)". There are two types of messages that TSM may send to HSEND. The first is a SEND_REQ message. This is a normal request for data from the host. How these messages are formated and sent will be discussed later in this chapter.

The other type of TSM request is a RETRY_REQ. The design of the asynchronous DIP allows for the possibility that on a busy CONVERSANT VIS with a slow host computer it is conceivable that a caller may hang up waiting for a host response. The next caller could inadvertantly get the previous caller's reply. To eliminate the chance of the caller getting the wrong data the protocol uses a timestamp on all messages. If the external function receives a timestamp other than the one it sent, it asks TSM to issue a RETRY_REQ. This puts the application back into a "waiting for response" mode. HSEND simply ignores the second request.

3.1.2.2 HREC Messages

Since HREC does not write to the TTY port directly it must ask the writer to do so on its behalf. Thus when the reader has need to acknowledge a response, it sends a message to HSEND asking it to write an ACK or a NAK to the TTY. HSEND complies and then goes back to sleep on its message queue.

HREC also communicates with HSEND to let it know when a request has been ACK'ed or NAK'ed. This is necessary since HSEND only writes to the port and does not read data. HREC uses a message type of "2" to identify these information-only messages. If HSEND sees one of these messages while it's asleep on its primary message receive, it discards it since it must be in response to an already timed-out request. This interaction will be discussed in further detail later.

3.1.2.3 Backup Writer Messages

The last source of possible messages is from the other writer process. This will only occur if the system has been configured in the Duplex mode and the backup writer is unable to use its own TTY port. The message appears the same as a request from TSM except that the morig field is set to the Queue Key of the backup writer. How the message is handled differently is discussed below.

3.1.3 Request Handling Procedure

Requests for data transmissions, from either TSM or the backup writer, are passed to the subroutine "build_head". This routine takes data passed to it from the voice script and TSM and combines them with formatting characters. The exact message format is as follows:

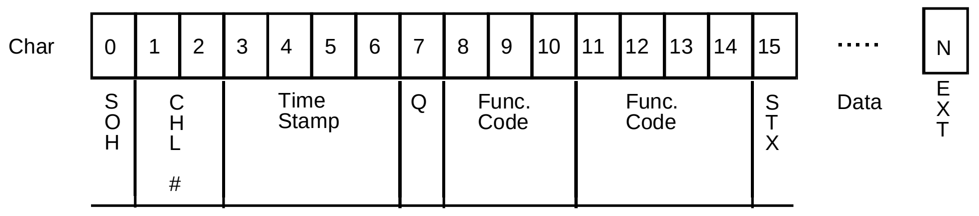


Figure 2.

Char 0 represents an ASCII Start Of Header (SOH) character. It is purely a formatting chararacter. The next two characters are the voice channel requesting the data. This is passed to HSEND from TSM via the header field mchan. It is sent to the host computer so it can be included in the response message and used by HREC to get the data back to the appropriate caller. The next four characters are the timestamp passed from the voice script. It has the format of MMSS. Character 7 is always a "Q" for outgoing messages.

The next three characters contain the function code passed in from the voice script. This function code can be used to route the request on the host to different server processes if desired. This field could be set to a default value in the application if the contents of the data is used to determine all required actions.

The next four characters represent a count of the number of characters passed in the data field (between the STX and the ETX). The routine does a strlen function on the character string passed to it from the voice script. An ASCII Start of Text (STX) follows the character count. The next characters are the actual data string passed from the voice script. Whether the data is fixed length fields or delimitted by a special character (such as a ",") is immaterial to the asynchronous DIP. What is important is that the creator of the data (the voice application writer) and the interpreter of the data (the host developer) come to an agreement on the context of the data field. The two restrictions that exist on the data is that it cannot be longer than 127 characters (a Script Builder field size limitation) and it cannot contain an ASCII End Of Text (EOT) character. The ETX used to signal the end of the message.

Once formatted, the data is then written to the TTY port. HSEND then sets a timer to go off at an interval of 1/4 the value passed in from the script. It then sleeps on its message queue waiting for an acknowledgment from HREC. At this point, HSEND is only looking for messages with the mtype field set to "2". This allows HSEND to re-send the request if required without worrying about getting a second request in the interim. If an ACK is received, HSEND returns to its main loop. If a NAK is received or a time-out occurs, HSEND re-transmits the request.

If the second request also fails, HSEND checks to see if the configuration is in the Duplex mode. If it is, it modifies the morig field to contain its Queue Key and then forwards the message to the backup writer. If the configuration is in the Simplex mode, HSEND returns the "999" error code back to the voice script.

Finally, if the request was received from the backup writer instead of TSM, the above scheme occurs with the following modifications. First, the 7th bit of the right-most character in the channel number field is set to one. This will allow HSEND when it gets the response back from the host to forward it back to TSM as if it were coming from the other writer process. The channel number is converted back to normal before the response is sent to TSM.

Second, if this is a backup request it will not be forwarded again after two failures. This keeps messages from being passsed back and forth forever.

3.2 The Reader Process (HREC)

This process is responsible for reading all the data that appears on the TTY port and for sending messages to HSEND or TSM as appropriate. Source code for HREC can be found in Appendix B.

3.2.1 Start Up Procedures

As HREC first starts up, it sets the signal catching routines and checks for proper syntax. Once that's completed, HREC determines whether it was invoked as hrec1 or hrec2. Based on its name, HREC gets the message queue number associated with the corresponding HSEND. After successfully getting the queue key, HREC initializes logging messages and then attempts to connect to the TTY port. If the open of the port is successful, HREC conditions the line much like HSEND did, except for reading instead of writing.

3.2.2 Transaction Handling Procedures

Once start up has completed, HREC goes into an infinite loop, sleeping on a read on the TTY port. Once data is read, control is transferred to the routine "get_resp". The first character of the data stream is examined. If it is an ACK or a NAK, HREC sends message with mtype set to "2" back to HSEND. If an ETX is received at this point, HREC assumes lost data and sends a normal request to HSEND to write out a NAK. If an SOH is read, HREC continues processing as outlined below. Any other characters are discarded and HREC returns to its primary loop.

On reading an SOH, HREC sets an alarm and then reads up to the STX. After that, HREC sets another alarm, reading until the ETX is encountered. If either timer go off, HREC sends a request to HSEND to write out a NAK on the TTY port. If all the data is received, HREC performs the following sanity checks. Failing any of these checks causes HREC to send a request to HSEND for a NAK.

1. validates the header; it must be the proper length and have either a "Q" or an "S" as the 7th character.
2. the size of the data identified in the header must agree with actual amount of data received between the STX and the ETX.
3. if the 7th character is a "Q" (request) it must have a function/response code of "002"

If the final test is true, HREC attempts to get the message queue key number of the "hook" DIP. If it succeeds, it sends the data to the "hook" DIP. This functionality is useful if there is data on the host that needs to be sent dynamically to the CONVERSANT VIS. The "hook" DIP can then take the data and parse it, then update a database record (for example). Note that it is the responsibilty of application "hook" developer to execute the VSstartup routine for the dynamic DIP named "hookdip". If a different name is desired, the source code for HREC would need to be modified and re-complied.

For regular responses, HREC sends a request to HSEND to write an ACK. HREC then examines the channel number to determine if the response should be sent to TSM as the primary or backup HSEND. The morig is set accordingly and the data is sent to TSM.

3.3 The External Function (HOSTREQ)

The final piece of custom software involved in the run time execution of the asynchronous DIP is the external function HOSTREQ. This function is written in native script and its listing can be found in Appendix C.

The function starts out by saving the timestamp information passed to it by the Script Builder voice application. It then sets the wait time for the dbase call equal to the value that was passed in from the voice script. It calculates roughly a fourth of that amount and puts that as well as the other info passed to it from the voice script into the structure HOST_REQ. The dbase call is then initiated using the dynamic name "async1" in HOSTREQ1 and "async2" in HOSTREC2.

If the dbase call returns a "-1" , HOSTREQ puts "xxxx" into the return character string and returns the "-1" to the script. If the dbase call times out, a "-2" is returned. HOSTREQ once again sets the return string to "xxxx" and returns the "-2" to the script. If the dbase call is successful, HOSTREQ examines the header to see if the returned timestamp matches the original one. If it does, the function/response code is extracted out of the header, converted to integer and returns.

If the timestamps do not match, HOSTREQ resets the time out value to half of what it was and issues another dbase call this time with the message type set to RETRY_REQ. It then handles responses the same as discussed above.

4. Installation Procedures

The asynchronous DIP is developed as an install package.

1. To install, type the command installpkg at the UNIX prompt. When instructed, place the DIP floppy into the drive.

   The voice system needs to be stopped when installing the asynchronous DIP package. If you have not done so prior to loading the package, it will prompt you before continuing. If you say no to the prompt, the installation will be aborted.

2. Once the voice system is stopped, you are prompted for the number host lines that you will be using. The choices are "1" or "2". If you respond by saying "1", you are asked for a request function number. The choices are "1" for HOSTREQ1 and "2" for HOSTREQ2. The choice really doesn't matter as long as you are consistent in your application.

If you say that you'll be using two lines, you are asked for the mode that you will be using for the DIP. The choices are "S" for simplex or "D" for duplex. In the simplex mode, the DIPs act independent of each other. Thus if half of your channels are using HOSTREQ1 and the TTY link associated with it is inoperable, then callers on those channels would get a "host down" condition. Under the duplex mode, if the HSEND1 fails twice, it attempts to use the other TTY port as described in "3. Design Considerations" above. The mode of operation can be changed after installation by using the command "asyncDIP_cf".

3. You are next prompted for the name(s) of the TTY port(s). It is only necessary to type in the tty portion of the name rather than the full path name. For example, use ttys04 rather than /dev/ttys04.

   Once that is completed, the install script continues, moving files to their appropriate locations. When finished, you are returned to the UNIX prompt. At that point you can re-start the voice system using the start_vs command.

4. To install the sample application, use the "cvis_menu" command to first add the application "async_test" and then to restore the speech and transaction parts of the application. Next, verify and install the application using Script Builder. Note that the application assumes the feature_tst package has been loaded onto the system already.

5. The stub or host simulator routine is installed into /vs/bin during the installpkg command. It can be run manually or set up to run automatically. To run it manually, type the command /vs/bin/hstub ttyXX where XX is the TTY number that is connected to the TTY port that HOSTREQ1 uses. The two ports must be connected using a null modem cable. This command can be run in the background by adding a "&" after the TTY number.

   To have the stub run automatically, use a file editor (such as vi) to add the following line to the file /etc/inittab:

```
AS9:respawn:/vs/bin/hstub ttyXX > /dev/null 2>&1
```

6. Write the file and then issue the command init q to activate the change. Note that changes will only stay in effect until the next time the voice system is stopped.

7. You can now assign the script "async_test" to a voice channel and begin testing.

5. Sample Application

This section covers the sample application "async_test" as well the simulated host program HSTUB. The application allows callers to enter 7 numbers and then choose to have the numbers reversed or doubled.

5.1 Async_test Script

The async_test application is written in Script Builder and its source listing can be found in Appendix D. The script begins by answering the phone and identifying itself . It then goes into an infinite loop that is terminated by the caller hanging up or by the use of too small of a time out value.

The script prepares for the message request by getting the current time and truncating it to the MMSS format. It also pre-pends the first two digits of the function code "00" to the header. It then prompts the call for any seven numbers. The script will not continue until it gets seven digits. Next the caller is prompted for a "1" to reverse the numbers or a "2" to double the numbers. The script will not continue until one of those two values is entered.

The script then completes the header and calls the HOSTREQ1 function. The script passes the header, the seven digits, a time out value, and the name of the character string that will receive the output data as arguments.

The script then responds with an appropriate message and then continues with the infinite loop. The only exception is in the case that the time out value in the HOSTREQ1 call was modified to a value less than 10 seconds. In that case, the script will terminate.

If the response is not what was expected, refer to "7. Asynchronous DIP Troubleshooting Procedures" for details on how to proceed.

5.2 Stub Routine

The stub routine is host simulation program written in "C" code. Its source listing can be found in Appendix E. The program looks and functions similar to HREC. It actually is nothing more than a modified version of HREC. It does the same setup for the reading of a TTY port, but it also opens a file pointer for writing as well. It does not have any DIPs directly associated with it and as such doesn't require any message queues or keys.

Once the header has been modified to the correct data length, the program will attempt to send the record up to two times.

6. Asynchronous DIP Modification Procedures

The Asynchronous Host Toolkit also installs a copy of the source code and make files necessary to generate the executables. If the directory /usr/async doesn't already exist, the installpkg command creates it. It then copies the C source code, custom header files, and the make file into that directory.

It is strongly recommended that the files be saved prior to making any changes. In the event of unrecoverable program changes and no backup of the files has occurred, the asynchronous DIP package can be removed using the removepkg command and then re-installed with the installpkg command. This procedure will remove the original source files located int the directory /usr/async and then restore the original version of the source code.

6.1 C Code Compilations

Once the code has been modified, HSEND and HREC can be recompiled using the following command make -f asyncdip.mk

This command will recompile both hsend.c and hrec.c. The programs can be compiled individually by adding the word hsend or hrec to the above command line. This make will create both versions of each program (i.e. HSEND1 and HSEND2) in the /usr/async directory. To test the new code, the current versions must be stopped. This can be accomplished by stopping the voice system through the stop_vs command or by editting the /etc/inittab file. To edit the inittab file, change the the line entries for the effected programs from "respawn" to "off" then write the file. Issue the command init q to turn off the processes. The new versions can then be copied to /vs/bin. The system can then be re-started using the start_vs command or the programs can be launched manually to test.

6.2 External Function Modifications

The external functions HOSTREQ1 and HOSTREQ2 reside in the directory /vs/bin/ag/lib as ".t" . The header files hostreq.h and hreqdef.h are there as well. Changes to the data structures in hostreq.h will require changes to HSEND and/or HREC. Note that another copy of hostreq.h exists in the directory /usr/async. These two header files must be kept current. If the data structures do need to change, in addition to recompiling the C code modules, run the command

```
mkheader hreq
```

using the file hostreq.h as the only input when prompted for an include file. This will create a new hreqdef.h file. This command creates an allocate program and its source code both of which can be deleted.

Once all the changes have been made, the voice application can be re-installed through Script Builder or by the command /vs/bin/ag/install <*application name*>.

For more detailed information on writing/modifying DIPs and external functions refer to *CONVERSANT VIS Version 4.0 Application Development*, 585-350-208.

7. Asynchronous DIP Troubleshooting Procedures

The Asynchronous Host Toolkit provides several methods for determining the nature of problems with the interface.

7.1 Logger Messages

There six conditions that will cause the asynchronous DIP to generate error messages to the Logger process. They are:

- ASYNC001 -- -- --- (ASYNC_QKEY) ASYNC DIP cannot get Qkey.
- ASYNC002 -- -- --- (ASYNC_CONN) ASYNC DIP cannot connect to port.
- ASYNC003 -- -- --- (ASYNC_WRITE) ASYNC DIP cannot write to port.
- ASYNC004 -- -- --- (ASYNC_READ) ASYNC DIP cannot read from port.
- ASYNC005 -- -- --- (ASYNC_UP) ASYNC Host link port is back up.
- ASYNC006 -- -- --- (ASYNC_DOWN) ASYNC Host link port appears to be down.

These messages, if generated, would show up when running the display error command or by generating the event report. More detail concerning the nature and possible recovery actions can be obtained by running the explain command with the error number or name (in parens above) as the argument.

7.2 Trace Messages

Much more data concerning the actual operations of the DIP can be obtained by using the CONVERSANT VIS trace command. The trace command will accept numerous processes as arguments to trace. To trace HSEND1/HREC1 use the argument "async1". To trace HSEND2/HREC2, use the argument "async2". In the duplex mode, both arguments can be used. A typical command might be trace tsm chan all async1 async2

The output from the above command can be redirected to a file or piped to a tee for later examination.

7.3 Printf Statements

Finally, since the asynchronous DIP comes as a toolkit with the source code, the "db_pr" trace statements can be globally changed to "printf" with more added as necessary. The entries in /etc/inittab could then be modified to send their output to a file. This would allow the programs to be running for some time, collecting data all the while. This might be particularly useful when trying to resolve intermittent problems. It is again strongly recommended that the code be saved with the "db_pr" trace statements left intact so that it can be restored after the trouble has been resolved.

HSEND Source Listing

A

```
/*  This is the Custom Services Group Asynchronous DIP, send process.  It can
be compiled singly or as part of a pair of DIPs, each providing service on a
separate async line for a different  external function.   *

 *      This process works in partnership with hrec.c

 *      Multiple instances of a single DIP are NOT supported.

 * MODULE NAME: hsend.c

 */

#include <stdio.h>

#include <sys/types.h>

#include <signal.h>

#include <ctype.h>

#include <errno.h>

#include <fcntl.h>

#include <termio.h>

#include <time.h>

#include "gdip_et.h"

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

#include "mesg.h"
```

```
#include "VS.h"

#include "spp.h"

#include "log.h"

#include "systemLog.h"

#include "logASYNC.h"

#include "dip_et.h"

#include "asyncdip.h"

#include "hostreq.h"

#define _INSTALLABLE_APPL 0

extern int errno;

char  wbuf[MAXSIZE];                    /* Buffer to async port */

int fd;                                 /* File descriptor */

int wfd;                                /* File descriptor */

int rc;

char dual_flag;                         /* D=dual DIPS, S=single DIP */

char procname[15];


char dev_name[12];

struct termio tty_attr;                 /* I/O line params */

struct termio save_attr;                /* I/O line params */


int alarm_val;                          /* Timer time_out value */

int alarm_cnt;                          /* Timer trap indicator */

int alarm_rtn();                        /* Timer trap routine */
int rsp;

int hstate;

int myQkey;             /* Queue key for the sender process */

int dualQkey;           /* Queue key for the 2nd sender */


struct host_reply reply;

struct host_req req;


main(argc, argv)

int argc;

char *argv[];

{

    int i;                  /* Temp counter */

    int cup();


    signal(SIGALRM, alarm_rtn);    /* Enable alarm */

    signal(SIGHUP,SIG_IGN);

    signal(SIGINT,cup);

    signal(SIGQUIT,cup);


    if(argc != 3){

        db_pr("SYNTAX: %s ttyXXX {D | S}>\n", argv[0]);

        exit(-1);

    }

    sprintf(dev_name,"/dev/%s",argv[1]);
```

```
        sprintf(procname,"%s",argv[0]);
        dual_flag = argv[2][0];
        hstate = 0;

        alarm_val = ALARM_VAL;          /* default time_out interval */
        setup();

        db_pr("%s: dev_name=%s, QKEY=%d\n",procname, dev_name, myQkey);

        while (1) {
          if((rc = mesgrcv(myQkey,&req,sizeof req, 0, 0,NULL)) < 0) {
              db_pr("%s: error reading queue, errno: (%d).\n",procname,errno);
              continue;
          }
          else{
               /* look for late ACK/NAK and discard if found */
               if(req.rqsthd.mtype == 2){
              db_pr("%s: discarding late ACK/NAK.\n",procname);
                     continue;
               }
/* message buffer header variable mcont contains the message request type */
              db_pr("%s: Got an event %d.\n",procname,req.rqsthd.mcont);
              if (req.timeout_val > 0){
                     if(dual_flag == 'D')
                    alarm_val = req.timeout_val;
```

```
      else
    alarm_val = req.timeout_val;
 }
 /* Determine type of request */
switch(req.rqsthd.mcont){
    case SEND_REQ: /* normal request */
        build_head();
        break;
    case REQ_4_ACK:/* send positive acknowledge */
        rsp = ACK;
        if ((i = dsend(wfd, &rsp, 1)) < 0){
     db_pr("%s: can't send ACK.\n",procname);
            sleep (10);
         }
        break;
    case REQ_4_NAK:/* send negative acknowledge */
        rsp = NAK;
        if ((i = dsend(wfd, &rsp, 1)) < 0){
     db_pr("%s: can't send NAK.\n",procname);
            sleep (10);
         }
        break;
    default:
        break;
}
```

```
            }


        }   /* end main while loop */


    }
    /****************************************************************
       * contains host channel path names
       * TTY attributes to change to and for saving original TTY atributes
     ****************************************************************/
    connect(device)
    char *device;
    {
            alarm (alarm_val);              /* Start timer */
           alarm_cnt = 0;                  /* Initialize indicator */

           if((wfd = open(device, O_WRONLY | O_NDELAY)) < 0){
               if (alarm_cnt == 0) {       /* check for time-out */
                   return(-1);
               } else {
                   return (-2);
               }
           }
            alarm (0);                    /* Kill timer */
           alarm_cnt = 0;                  /* Reset indicator */
    /* * copy original attributes and setup raw mode */
```

```
        if(ioctl(wfd, TCGETA, &tty_attr) == -1) {
             db_pr ("%s: Connect: ioctl get failed, errno=%d\n",procname,errno);
            return (-3);
        }
        if(ioctl(wfd, TCGETA, &save_attr) == -1) {
             db_pr ("%s: Connect: ioctl get failed, errno=%d\n",procname,errno);
            return (-3);
        }
        tty_attr.c_iflag = (IGNPAR | IGNBRK | ISTRIP | IXON | IXOFF);
        tty_attr.c_oflag = 0;
        tty_attr.c_cflag = (CREAD | B9600 | CS8 | CLOCAL );
        tty_attr.c_lflag &= ~ICANON;
        tty_attr.c_lflag &= ~ECHO;
        tty_attr.c_cc[4] = 0;
        tty_attr.c_cc[5] = 30;
        if(ioctl(wfd, TCFLSH, 1) == -1) {
             db_pr ("%s: Connect: ioctl flush failed, errno=%d\n",procname,errno);
            return (-3);
        }
        if(ioctl(wfd, TCSETA, &tty_attr) == -1) {
             db_pr ("%s: Connect: ioctl set failed, errno=%d\n",procname,errno);
            return (-3);
        }
        return(wfd);
}
```

```
/* this routine does the writing of information out to the TTY port */

dsend(sfd, data, size)

        char *data;                    /* address of send buffer */

        int size;                      /* size of buffer in bytes */

        int sfd;                       /* file descriptor */

{

        int i;


        i = write(sfd, data, size);

        if (i < 0) {

                db_pr ("%s: Error writing: return = %d, errno =
%d\n",procname,i,errno);

                logMsg(ASYNC_WRITE,EL_FL,dev_name,errno);

        }

        db_pr("%s: sent message, length = %d\n",procname,i);


        return(i);

}

        /* Alarm handling routine - incrments counter to let

           other functions know that a time-out has occured */

alarm_rtn(sig_num)

        int sig_num;

{

        (void)signal(SIGALRM, alarm_rtn);

        db_pr("%s: got alarm signal\n",procname);

        alarm_cnt++;
        return (0);

}

setup()

{

        /* Determine if this is the primary or secondary sender and get
corresponding message queue */

        if(strcmp(procname,"/vs/bin/hsend2") == 0){

                myQkey = VSstartup("async2",0, DIP_PROC);

                logInit("ASYNC2");

        }

        else{

                myQkey = VSstartup("async1",0, DIP_PROC);

                logInit("ASYNC1");

        }

        /* If unable to get qkey, log error message and quit */

        if(myQkey <= 0){

                logMsg(ASYNC_QKEY,EL_FL);

               sleep(ALARM_VAL);

               exit(-1);

        }

        /* Initial tracing capability */

        db_init(myQkey);

        db_pr("%s: attempting connect to %s\n",procname,dev_name);

        if ((fd = connect(dev_name)) < 0) {

            db_pr("%s: Async DIP unable to connect to %s\n",procname,dev_name);

            logMsg(ASYNC_CONN,EL_FL,dev_name);
            exit(-1);
        }
        /* get Qkey of other sender if duplex mode */
        if( dual_flag == 'D'){
                if(strcmp(procname,"/vs/bin/hsend2") == 0)
                        dualQkey = VStoqkey("async1");
                else
                        dualQkey = VStoqkey("async2");
        }
        /* sleep to allow reader time to start up */
        sleep(ALARM_VAL);
}
/*****************************************************************************************
*This routine builds the message based on data passed in fromthe script as well
as internal formatting rules.
*****************************************************************************************/
build_head()
{
        int     i,slen,hlen,retry;
        char    etx, datestamp[5],ohdr[17];
        char    soh,stx,msg_id[8],data_len[5],func_code[4];


        soh = SOH;
        stx = STX;
        etx = ETX;
        slen = strlen(req.input_val);
```

```
/* get data from header passed from script */

sscanf(req.inheader,"%4s%3s",datestamp,func_code);
sprintf(msg_id,"%2.2d%4s",req.rqsthd.mchan,datestamp);

/* Dual-DIP request encoding:
 *   Bit 7 of the right-most char of the channel field of the
 *   protocol header indicates DUAL (bit on) or not (bit off).
 *   If the message came from the alternate DIP rather than TSM,
 *   then turn the bit on so that the receive process can then
 *   send the reply to TSM as if it came from the alternate DIP.
 */
if (req.rqsthd.morig == dualQkey)
    msg_id[1] = msg_id[1] | DUAL_SET;

sprintf(data_len,"%4d",slen);

/* build outgoing header */
sprintf(ohdr,"%c%6sQ%3s%4s%c",soh,msg_id,func_code,data_len,stx);
hlen = strlen(ohdr);

/* attempt to send data to TTY port up to two times */
retry = 0;
while(retry < 2){
        if ((i = dsend(wfd, ohdr, hlen )) < 0)

            sleep (ALARM_VAL);

        db_pr("%s: input = %s\n",procname,req.input_val);

        if ((i = dsend(wfd, req.input_val, slen )) < 0)

            sleep (ALARM_VAL);

        if ((i = dsend(wfd, &etx, 1)) < 0)

            sleep (ALARM_VAL);

/* Now wait for reply from send to host.  This comes via the hrec process and the
response msg queue. */

        db_pr("%s: Time out value: %d\n",procname,alarm_val);

        rc = alarm (alarm_val);             /* Start timer */

        db_pr("%s: alarm return value: %d\n",procname,rc);

        alarm_cnt = 0;                 /* Initialize indicator */

        sleep(1);


        mesgrcv(myQkey,&req, sizeof req, 2, 0, NULL);


        if (alarm_cnt != 0){

            db_pr("%s: Time out on ack\n",procname);

            alarm(0);

            retry++;

            continue;

        }

      /* turn off and reset timer */

        alarm(0);

        alarm_cnt = 0;
```

```
          /* check status of reply */
          switch(req.rqsthd.mcont){
               case GOT_ACK:   db_pr("%s: received ACK\n",procname);
                                 if(hstate == 1){
                                      logMsg(ASYNC_UP,EL_FL,dev_name);
                                      hstate = 0;
                                 }
                               return(1);
                               break;;
               case GOT_NAK:   retry++;
                               db_pr("%s: received NAK\n",procname);
                                 if(hstate == 1){
                                      logMsg(ASYNC_UP,EL_FL,dev_name);
                                      hstate = 0;
                                 }
                               break;;
               default:          db_pr("%s: got %c\n",procname,req.rqsthd.mcont);
                               retry++;
                               break;;
          }
     }
/* Now send the request to the other DIP in the pair, but only if
     this is a dual-DIP arrangement and the current request came
     from TSM, not the other DIP in the pair. */
if ((dual_flag == 'D') && (req.rqsthd.morig == TSM)) {
```

```
                /* if the attempt to get the other DIP's Qkey failed

                   during setup, get it now */


                if(dualQkey < 0){

                      if( dual_flag == 'D'){

                            if(strcmp(procname,"/vs/bin/hsend2") == 0)

                                  dualQkey = VStoqkey("async1");

                            else

                                  dualQkey = VStoqkey("async2");

                      }

                }

                db_pr("%s: Primary hsend failure, requesting dual DIP
send\n",procname);

               req.rqsthd.morig = myQkey;

               if (mesgsnd(dualQkey,&req,sizeof req,0) < 0)

                    db_pr("%s: Cannot send message to duplex DIP\n",procname);

               else {

                    return(0);

                }

         }

/* If I got to this point, it means I was unable to send to the host (or to 2nd DIP in
a dual-DIP pair), thus I need to tell the script */

          strcpy(reply.outheader, req.inheader);

         strcpy(reply.output_val, req.input_val);

         reply.outheader[4] = '9';

         reply.outheader[5] = '9';
```

```
        reply.outheader[6] = '9';

        db_pr("%s: No host response, header = %s\n",procname,reply.outheader);

        db_pr("%s: data = %s\n",procname,reply.output_val);

        reply.rplyhd = req.rqsthd;

/*  The morig value ensures reply goes to the correct script */

        if (reply.rplyhd.morig == TSM)

                reply.rplyhd.morig = myQkey;


        if(mesgsnd(TSM,&reply,sizeof reply,0) < 0)

                db_pr("%s: Cannot send message to TSM\n",procname);


        if(hstate == 0){

                logMsg(ASYNC_DOWN,EL_FL,dev_name);

                hstate = 1;

        }


}

/*      signal catching clean up routine  */

cup(sig)

int     sig;

{

int     i;

        sprintf(wbuf,"\004\004\004\004\004");

                if ((i = dsend(wfd, wbuf, 5 )) < 0)

                        sleep (RCV_ALARM);
```

```
          if(ioctl(wfd, TCFLSH, 1) == -1) {
                db_pr ("%s: Connect: ioctl flush failed, errno=%d\n",procname,errno);
                 return (-3);
          }
          if(ioctl(wfd, TCSETA, &save_attr) == -1) {
                db_pr ("%s: ioctl reset failed, errno=%d\n",procname,errno);
          }
          close(wfd);
          db_pr("%s: Received signal(%d) - exitting.\n",procname,sig);
          exit(0);
}
/* function for determining the first log message for the ASYNC message class */
Fcn_ASYNCMSG_START()
{
          return(logStartClass("ASYNC"));
}
```

HREC Source Listing

B

```
/*  This is the Custom Services Group Asynchronous DIP, receive process.
 *     It can be compiled singly or as part of a pair of DIPs, each
 *     providing service on a separate async line for a different
 *     external function.
 *
 *     This process works in partnership with hsend.c
 *     Multiple instances of a single DIP are supported.
 * MODULE NAME: hrec.c
 */
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <termio.h>
#include <time.h>
#include "gdip_et.h"
#include <sys/ipc.h>
```

```
#include <sys/msg.h>

#include <string.h>

#include "mesg.h"

#include "VS.h"

#include "spp.h"

#include "log.h"

#include "systemLog.h"

#include "logASYNC.h"

#include "dip_et.h"

#include "asyncdip.h"

#include "hostreq.h"

#define _INSTALLABLE_APPL 0


extern int errno;


char  nbuf[MAXSIZE+2];                  /* Buffer from async port */

char *rptr;

int fd;                                 /* File descriptor */

int rc;

char dual_flag;                         /* D=dual DIPS, S=single DIP */

char procname[15];


int myQkey;              /* Queue key for this DIP */

int dualQkey;            /* Queue key for alternate sender DIP */

int hookQkey;            /* Queue key for unsolicited msg DIP */
```

```
char dev_name[12];
struct termio tty_attr;              /* I/O line params */
struct termio save_attr;             /* I/O line params */

int alarm_cnt;                       /* Timer trap indicator */
int alarm_rtn();                     /* Timer trap routine */

struct host_reply reply;
struct host_req req;

main(argc, argv)
int argc;
char *argv[];
{
    int i;                           /* Temp counters */
    int cup();

    signal(SIGALRM, alarm_rtn);      /* Enable alarm */
    signal(SIGHUP,SIG_IGN);
    signal(SIGINT,cup);
    signal(SIGQUIT,cup);

    if(argc != 2){
        db_pr("SYNTAX: %s ttyXXX \n", argv[0]);
```

```
                exit(-1);
            }

            sprintf(dev_name,"/dev/%s",argv[1]);
            sprintf(procname,"%s",argv[0]);

            setup();

            db_pr("%s: dev_name=%s, QKEY=%d\n",procname, dev_name, myQkey);

            /* loop forever, looking for data on the TTY port */
            while (1) {
                if ((i = receive(fd, nbuf, 1)) < 0) {
                    sleep (RCV_ALARM);
                    continue;
                }

                /* if data found, process it */
                if(i > 0){
                    get_resp();
                }

            }

        }
```

```
/*******************************************************************
   * contains host channel path names
   * TTY attributes to change to and for saving original TTY atributes
 *******************************************************************/

connect(device)
char *device;
{
        int fd1;

        alarm (ALARM_VAL);              /* Start timer */
        alarm_cnt = 0;                  /* Initialize indicator */

        if((fd1 = open(device, O_RDONLY )) < 0){
         db_pr("%s: Connect: open dev_name=<%s> failed,
errno=%d\n",procname,device, errno);
            if (alarm_cnt == 0) {       /* check for time-out */
                return(-1);
            } else {
         db_pr("%s: Time-out on read open; possibly try later\n",procname);
                return (-2);
            }
        }
```

```
alarm (0);                    /* Kill timer */
alarm_cnt = 0;                  /* Reset indicator */


/*
 * copy original attributes and setup raw mode
*/
if(ioctl(fd1, TCGETA, &tty_attr) == -1) {
    db_pr ("%s: Connect: ioctl get failed, errno=%d\n",procname,errno);
   return (-3);
}
if(ioctl(fd1, TCGETA, &save_attr) == -1) {
    db_pr ("%s: Connect: ioctl get failed, errno=%d\n",procname,errno);
   return (-3);
}
tty_attr.c_iflag = (IGNPAR | IGNBRK | ISTRIP | IXON | IXOFF);
tty_attr.c_oflag = 0;
tty_attr.c_cflag = (CREAD | B9600 | CS8 | CLOCAL );
tty_attr.c_lflag &= ~ICANON;
tty_attr.c_lflag &= ~ECHO;
tty_attr.c_cc[4] = 1;/* wait for at least 1 character */
tty_attr.c_cc[5] = 90;/* allow 9 seconds between chars */
if(ioctl(fd1, TCFLSH, 0) == -1) {
    db_pr ("%s: Connect: ioctl flush failed, errno=%d\n",procname,errno);
   return (-3);
}
```

```
        if(ioctl(fd1, TCSETA, &tty_attr) == -1) {
            db_pr ("%s: Connect: ioctl flush failed, errno=%d\n",procname,errno);
            return (-3);
        }

        return(fd1);
}

/*****************************************************************
 * It is called to receive maximum of 'size' bytes of data from the
 * host computer. The process will be roadblocked until the requested
 * amount of data is return by the host. The caller must pass a pointer
 * to the buffer where the requested data should be stored.
 *
 * RET:
 * n, where n >= 0. Returns the actual byte count of received data.
 * FAILED, if fails for some other reasons.
 *
 * IMPLEMENT:
 * - call read(),
 * - check return value.
 *****************************************************************/
receive(fd, data, size)
        char *data;                 /* address of recieve buffer */
        int size;                   /* size of buffer in bytes */
        int fd;                 /* file descriptor */
{
        int i;


        i = read(fd, data, size);
        if (i < 0) {
                logMsg(ASYNC_READ,EL_FL,dev_name,errno);
        }
        return(i);
}


/*      Alarm handling routine - increments counter to let other
        routines know that a time-out has occured */
alarm_rtn(sig_num)
        int sig_num;
{
        (void)signal(SIGALRM, alarm_rtn);
        alarm_cnt++;
        return (0);
}


setup()
{


        /* Determine if this is the primary or secondary sender
```

```
   and get the coresponding message queue */

 if(strcmp(procname,"/vs/bin/hrec2") == 0){

         myQkey = VStoqkey("async2");

         logInit("ASYNC2");

 }

 else{

         myQkey = VStoqkey("async1");

         logInit("ASYNC1");

 }



 /* If unable to get qkey, log error message and quit */

  if(myQkey <= 0){

         logMsg(ASYNC_QKEY,EL_FL);

        sleep(ALARM_VAL);

        exit(-1);

}


 /* Initial tracing capability */

 db_init(myQkey);


db_pr("%s: attempting connect to %s\n",procname,dev_name);

if ((fd = connect(dev_name)) < 0) {

     db_pr("%s: Async DIP unable to connect to %s\n",procname,dev_name);

    logMsg(ASYNC_CONN,EL_FL,dev_name);
```

```
                exit(-1);
            }

    }

    /*******************************************************************
    *
    *       This routine determines the type and validity of data received
    *       on the TTY port. It formats requests to the sender process and
    *       TSM as appropriate.
    *
    *******************************************************************/

    get_resp()
    {
            int     dlen,hdlen,hlen,ichan,sflag,j,i;
            char    *dptr,*rptr,*optr,nextchar,rhead[20];
            char    tstamp[5],hclen[6],rdata[MAXSIZE],qhead[8],rchan[3],fcode[4];

            /* set default message parameters */
            req.rqsthd.mtype = 0;
            req.rqsthd.mcont = REQ_4_NAK;

            /* look at first character in buffer */
            switch(nbuf[0]){
```

```
case ACK:
            /* got positive acknowledgement, let sender
              know. Use mtype 2 to give message special
              consideration. */
      req.rqsthd.mcont = GOT_ACK;
      req.rqsthd.mtype = 2;
          mesgsnd(myQkey,&req,sizeof req, 0);
          return(1);
          break;
case NAK:

            /* got negative acknowledgement, let sender
              know. Use mtype 2 to give message special
              consideration. */
      req.rqsthd.mcont = GOT_NAK;
      req.rqsthd.mtype = 2;
          mesgsnd(myQkey,&req,sizeof req, 0);
          return(1);
          break;;
case SOH:/* start of good reply data */
          break;;
case ETX:      /* ETX without SOH - error */
          req.rqsthd.mcont = REQ_4_NAK;
          mesgsnd(myQkey,&req, sizeof req, 0);
          return(-1);
```

```
                        break;;
                default:        /* all other data - discard */
                        return(1);
                        break;;
        }

        alarm (RCV_ALARM);              /* Start timer */
        alarm_cnt = 0;                  /* Initialize indicator */
        sflag = 0;
        j = 0;

        /* read header data to STX or until time-out occurs */
        while ((alarm_cnt == 0) && ( sflag == 0)){     /* check for time-out */
            if ((i = receive(fd, &nextchar, 1)) < 0)
                continue;

            if(i > 0){
                switch(nextchar){
                        case STX:       sflag = 1;
                                        break;;
                        default:        rhead[j++] = nextchar;
                                        break;;
                }
                continue;
            }
```

```
}
if (alarm_cnt != 0){
        db_pr("%s: Time out on rest of header\n",procname);
        mesgsnd(myQkey,&req, sizeof req, 0);
        return(-1);
}

/* turn off and reset timer */
alarm(0);
alarm_cnt = 0;

rhead[j] = '\0';
db_pr("%s: got header %s\n",procname,rhead);

alarm (RCV_ALARM);            /* Start timer */
alarm_cnt = 0;                /* Initialize indicator */
sflag = 0;
j = 0;

/* get the rest of the data to ETX or time-out occurs */
while ((alarm_cnt == 0) && ( sflag == 0)){     /* check for time-out */
    if ((i = receive(fd, &nextchar, 1)) < 0)
        continue;

    if(i > 0){
            switch(nextchar){
                case ETX:      sflag = 1;
                               break;;
                default:       rdata[j++] = nextchar;
                               break;;
             }
             continue;
         }
      }
      if (alarm_cnt != 0){
            db_pr("%s: Time out on data\n",procname);
            mesgsnd(myQkey,&req, sizeof req, 0);
            return(-1);
      }

       /* turn off and reset alarm */
      alarm(0);
      alarm_cnt = 0;

      rdata[j] = '\0';
       db_pr("%s: got data %s\n",procname,rdata);

       /* validate header */
      hlen = strlen(rhead);
      if((hlen != 14) || ((rhead[6] != 'S') && (rhead[6] != 'Q'))){
        db_pr("%s: garbled header received\n",procname);
        mesgsnd(myQkey,&req, sizeof req, 0);
        return(-1);
}


/* get channel number */
rptr = rhead;
strncpy(rchan,rptr,2);
rchan[2] = '\0';


/* Dual-DIP request decoding:
 *   Bit 7 of the right-most char of the channel field of the
 *   protocol header indicates DUAL (bit on) or not (bit off).
 *   If the message originated from the alternate DIP, then turn
 *   the bit off to restore the channel number digit to its
 *   proper value and set the dual flag so that the reply will
 *   go to TSM as if it came from the alternate DIP.
 */
if ((rchan[1] & DUAL_SET) > 0) {
        rchan[1] = rchan[1] & DUAL_UNSET;
        dual_flag = 'D';
        if(strcmp(procname,"/vs/bin/hrec2") == 0)
                dualQkey = VStoqkey("async1");
        else
                dualQkey = VStoqkey("async2");
```

```
}
else
        dual_flag = 'S';

ichan = atoi(rchan);

/* extract time stamp */
rptr+=2;
strncpy(tstamp,rptr,4);
tstamp[4] = '\0';

/* extract response code */
rptr+=5;
strncpy(fcode,rptr,3);
fcode[3] = '\0';

/* extract data length from header */
rptr+=3;
strcpy(hclen,rptr);

sprintf(qhead,"%s%s",tstamp,fcode);

hdlen = atoi(hclen);
dlen = strlen(rdata);
```

```
/* reject all messages that are reQuests with function/response
   code other than 002 */
if((rhead[6] == 'Q') && (strcmp(fcode,"002") != 0)) {
     db_pr("%s: invalid data received\n",procname);
     mesgsnd(myQkey,&req, sizeof req, 0);
     return(-1);
}

/* compare what the host thinks the data length is versus what was
   actually received */
if(hdlen != dlen){
     db_pr("%s: garbled data received; header said %d, data was
%d\n",procname,hdlen,dlen);
     mesgsnd(myQkey,&req, sizeof req, 0);
     return(-1);
}

/* data looks good, let sender know */
req.rqsthd.mcont = REQ_4_ACK;
mesgsnd(myQkey,&req, sizeof req, 0);

/* build response to TSM or hook DIP */
strcpy(reply.outheader, qhead);
strcpy(reply.output_val, rdata);

reply.rplyhd.mcont = 0;
```

```
        reply.rplyhd.mtype = 1;
        if (dual_flag == 'D')
        reply.rplyhd.morig = dualQkey;
        else
        reply.rplyhd.morig = myQkey;
        reply.rplyhd.mchan = ichan;

        /* check for special host message */
        if((strcmp(fcode,"002") == 0) && (rhead[6] == 'Q')) {
                db_pr("%s: got unsolicited message\n",procname);
                 hookQkey = VStoqkey("hookdip");
                if(mesgsnd(hookQkey,&reply, sizeof reply, 0) < 0)
                        db_pr("%s: Cannot send message to hook DIP\n",procname);
        }
        else
                if(mesgsnd(TSM,&reply,sizeof reply,0) < 0)
                        db_pr("%s: Cannot send message to TSM\n",procname);
}

/*      signal handling clean up routine */
cup(sig)
int     sig;
{
int     i;
```

```
        if(ioctl(fd, TCFLSH, 0) == -1) {
                db_pr ("%s: Connect: ioctl flush failed, errno=%d\n",procname,errno);
            return (-3);
        }
        if(ioctl(fd, TCSETA, &save_attr) == -1) {
                db_pr ("%s: ioctl reset failed, errno=%d\n",procname,errno);
        }
        close(fd);
        db_pr("%s: Received signal(%d) - exitting.\n",procname,sig);
        exit(0);
}


/* function to determine first log message index to be used by the ASYNC
   message class */
Fcn_ASYNCMSG_START()
{
        return(logStartClass("ASYNC"));
}
```

HOSTREQ Source Listing

C

```
/*
*FUNCTION: hostreq1     (for 1st set of channels)
*
*INPUT:
*       ARG1:   header; format - mmssfff
*               mm = minutes after the hour
*               ss = seconds after the hour
*               fff - function code
*
*       ARG2:   variable function specific data, fields
*               separated by commas
*
*       ARG3:   async interface alarm interval
*               numeric = seconds for this function to wait for response
*
*OUTPUT:
*
*       ARG4:   variable function specific data, fields
*           separated by commas
*
*     RVAL:   response code
*
*REMARKS:    This function will try twice to get a response that
*           corresponds to its request.  On retry, the DIP will
*           throw away the request, assuming that the bad response
*           (wrong time in header) was an old one from a prior
*           request that failed.
*
*           The alarm interval calculated for the DIP is based on
*           there being two DIPs, each allowing for 2 tries at
*           successfully sending a request
*/

DEFSPACE(stime,8)
DEFSPACE(rtime,8)
DEFARG_COUNT(4)
DEFARG(inhead,char,in)
DEFARG(inval,char,in)
DEFARG(timeout,num,in)
DEFARG(outval,char,out)

#include "hostreq.h"
#include "hostreq1.h"
```

```
#include "hreqdef.h"


L__hostreq1:
          load(r.0, imm.-1)          /* prepare for invalid dbase call */


          strcpy(ch.F_stime,*ch.3)        /* save request send time */
          load(r.1,imm.F_stime)
          incr(r.1,imm.4)
          load(*ch.1,imm.0)
          trace(imm.9500,ch.F_stime)


          strcpy(ch.INHEADER,*ch.3)
          strcpy(ch.INPUT_VAL,*ch.2)
           load(r.1,int.F__FUNCT_ARG3)
          jmp (r.1 < imm.10 ,L__badarg1)


          nwitime(r.1)                    /* set time_out for dbase call*/
          decr(r.1,imm.2)                   /*  set time for DIP to be:   */
          div(r.1,imm.4)                  /*    (nwitime - 2) / 4       */
          load(int.TIMEOUT_VAL,r.1)
           trace(imm.9512,int.TIMEOUT_VAL)


          /* The m_cont (msg type) value tells the DIP that this is */
          /*    a host request from a transaction external function */
```

```
        dbase(im."async1", SEND_REQ, int.HOST_REPLY, SZHOST_REPLY,
int.HOST_REQ, SZHOST_REQ)

        jmp (r.0 < imm.0 L__dfail1 )

        load(r.1, imm.F_rtime)          /* save request reply time */

        strcpy(*ch.1, ch.OUTHEADER)

        incr(r.1,imm.4)

        load(*ch.1,imm.0)

        trace(imm.9501,ch.F_rtime)


        strcmp(ch.F_stime,ch.F_rtime)   /*  see if times are same   */

        jmp (r.0 != imm.0 L__retry1)     /*      else request failed */


        load(r.1, int.F__FUNCT_ARG4)    /* retrieve output for client */

        strcpy(*ch.1, ch.OUTPUT_VAL)

        load(r.1, imm.OUTHEADER)

        incr(r.1,imm.4)

        atoi(r.0,*ch.1)

        rts()


L__badarg1:

        load(r.0,imm.-3)/* too small a time-out value */

L__dfail1:

        strcpy(ch.F__FUNCT_ARG4, imm."xxxx")    /* mark bad return */

        rts()


L__retry1:
        load(r.1,int.F__FUNCT_ARG3)

        div(r.1,imm.2)                  /*   wait less time on retry */

        nwitime(r.1)

        dbase(im."async1", RETRY_REQ, int.HOST_REPLY, SZHOST_REPLY,
int.HOST_REQ, SZHOST_REQ)

        jmp (r.0 < imm.0 L__dfail1 )


        load(r.1, int.F__FUNCT_ARG4)    /* retrieve output for client */

        strcpy(*ch.1, ch.OUTPUT_VAL)

        load(r.1, imm.OUTHEADER)

        incr(r.1,imm.4)

        atoi(r.0,*ch.1)

        rts()
```

Test Script

D

```
start:
1.  Answer Phone
2.  Announce
      Speak With Interrupt
         Phrase: "This is the asynchronous data interface program te"
         Phrase: "sil.500"
     get_number:


     #This is start of the main loop. It gets the
     #current time and parses it for use in the header.


3.  External Function
      Function Name: u_datetime
      Use Arguments: todays_date  current_time  $UNIX_TIME
4.  External Function
      Function Name: substring
      Use Arguments: subtime  current_time  2  4
5.  External Function
```

```
        Function Name: concat
        Use Arguments: subheader  subtime  "00"  0


      #input data prompt - it loops until 7 digits are
      #entered.


   6.  Prompt & Collect
        Prompt
          Speak With Interrupt
            Phrase: "Please enter any seven digits."
            Phrase: "sil.500"
   Input
          Caller Input Field: in_digits
          Min Number Of Digits: 07
          Max Number Of Digits: 07
        Checklist
          Case: "Input Ok"
            Continue
          Case: "Initial Timeout"
            Reprompt
          Case: "Too Few Digits"
            Reprompt

          Case: "No More Tries"
            Goto get_number
```

```
End Prompt & Collect

get_choice:


#This is the start of the choice loop. Only by

#entering a 1 or a 2 will the script continue.


7. Prompt & Collect

    Prompt

      Speak With Interrupt

        Phrase: "To reverse the digits, please press 1."

        Phrase: "sil.500"

        Phrase: "To double the digits, please press 2."

        Phrase: "sil.500"

    Input

      Caller Input Field: func_code

      Max Number Of Digits: 01

    Checklist

      Case: "1"

        Continue

      Case: "2"

        Continue

      Case: "Not On List"

        Reprompt

      Case: "Initial Timeout"

        Reprompt
```

```
            Case: "Too Few Digits"
              Reprompt
            Case: "No More Tries"
              Goto get_choice
        End Prompt & Collect


        #complete the header by appending the choice to the
        #function code.


     8. External Function
          Function Name: concat
          Use Arguments: header  subheader  func_code  0


        #This is the async DIP external function. By using
        #hostreq1, it expects hsend1 and hrec1 to be
        #available. It sends the constructed header, the
        #input data, and the time out value to the process
        #hsend1. The returned data is placed in the outdata
        #variable and the response code is placed in the
        #variable return_val.


     9. External Function
          Function Name: hostreq1
          Use Arguments: header  in_digits  20  out_data
          Return Field:  return_val
```

```
10. Evaluate
    If return_val   = -1

      #internal problems - see error logger or trace

11.    Announce
          Speak Without Interrupt
            Phrase: "I'm sorry, but there are internal systems problems"
            Phrase: "sil.500"
    Elseif return_val   = -2

      #Time out; ACK received but no response in time

12.    Announce
          Speak Without Interrupt
            Phrase: "I'm sorry, but there is no response from the host."
            Phrase: "sil.500"
    Elseif return_val   = -3

      #time out parameter < 10

13.    Announce
          Speak Without Interrupt
            Phrase: "I'm sorry, but the selected time-out value is too"
            Phrase: "sil.500"
```

```
14.   Disconnect

15.   Quit

   Elseif return_val   = 999

     #No host response; not even ACK/NAK

16.   Announce

       Speak Without Interrupt

         Phrase: "I'm sorry, but there is no connection to the host."

         Phrase: "sil.500"

   Elseif return_val   = 1

     #valid response - for response code 1

17.   Announce

       Speak Without Interrupt

         Phrase: "The reverse of the digits is"

         Field: out_data As C

         Phrase: "sil.500"

   Elseif return_val   = 2

     #valid response - for response code 2

18.   Announce

       Speak Without Interrupt
```

```
        Phrase: "Double the digits is"

        Field: out_data As C

        Phrase: "sil.500"

    End Evaluate


    #go back to the start of the main loop


19.  Goto get_number
```

APPENDIX E - Stub Program

```
/*  This is a stub program that simulates a host for testing the send and receive
parts of the CSG asynchronous DIP.

 *

 *     This process works in partnership with hsend.c and hrec.c

 * MODULE NAME: hstub.c

 */

#include <stdio.h>

#include <sys/types.h>

#include <signal.h>

#include <ctype.h>

#include <errno.h>

#include <fcntl.h>

#include <termio.h>

#include <sys/ipc.h>

#include <sys/msg.h>
```

```
#include <string.h>
#include "mesg.h"
#include "dip_et.h"
#include "asyncdip.h"
#include "hostreq.h"

extern int errno;

char  nbuf[MAXSIZE+2];              /* Buffer from async port */
int fd;                        /* File descriptor */
int wfd;                       /* File descriptor */
char dev_name[12];
struct termio tty_attr;             /* I/O line params */
struct termio save_attr;             /* I/O line params */

int alarm_val;                 /* Timer time_out value */
int alarm_cnt;                 /* Timer trap indicator */
int alarm_rtn();               /* Timer trap routine */

main(argc, argv)
int argc;
char *argv[];
{
     int i;             /* Temp counter */
     int cup();
```

```
signal(SIGALRM, alarm_rtn);     /* Enable alarm */
signal(SIGHUP,SIG_IGN);
signal(SIGINT,cup);
signal(SIGQUIT,cup);

if(argc != 2){
        printf("\nSYNTAX: hstub ttyXXX\n");
         fflush(stdout);
        exit(-1);
}
 sprintf(dev_name,"/dev/%s",argv[1]);

 alarm_val = ALARM_VAL;/* default time_out interval */
setup();
 printf("\nfd(read)=%d, fd(write)=%d\n", fd, wfd);
 fflush(stdout);

 /* loop forever, looking for data on TTY port */
while(1){
        if ((i = receive(fd, nbuf, 1)) < 0) {
                 fflush(stdout);
                exit(-1);
        }
         /* if error or no data, wait and then try again */
```

```
                if(i < 1) {
                      sleep (2);
                     continue;
                 }
                 /* process incoming data */
                 get_resp();
                 fflush(stdout);
        }/* end main while loop */
}

 /******************************************************************
  * contains path name of line from the DIP to this simulated host
  * TTY attributes to change to and for saving original TTY atributes
 *****************************************************************/

connect(device)
char *device;
{
        int fd1;

        alarm (alarm_val);              /* Start timer */
        alarm_cnt = 0;                  /* Initialize indicator */

        if((fd1 = open(device, O_RDONLY)) < 0){
                 printf ("Connect: open dev_name=<%s> failed, errno=%d\n",device,
errno);
```

```
        if (alarm_cnt == 0) {      /* check for time-out */
            return(-1);
        } else {
             printf ("Time-out on read open; possibly try later\n");
            return (-2);
        }
    }
    if((wfd = open(device, O_WRONLY | O_NDELAY)) < 0){
             printf ("Connect: open dev_name=<%s> failed, errno=%d\n",device,
errno);
        if (alarm_cnt == 0) {      /* check for time-out */
            return(-1);
        } else {
             printf ("Time-out on write open; possibly try later\n");
            return (-2);
        }
    }


    printf ("Open call succeeded, fd=%d\n",fd1);
    alarm (0);                    /* Kill timer */
    alarm_cnt = 0;                  /* Reset indicator */
    /*
     * copy original attributes and setup raw mode
    */
    if(ioctl(fd1, TCGETA, &tty_attr) == -1) {
        printf ("Connect: ioctl get failed, errno=%d\n",errno);
    return (-3);
}
if(ioctl(fd1, TCGETA, &save_attr) == -1) {
     printf ("Connect: ioctl get failed, errno=%d\n",errno);
    return (-3);
}
tty_attr.c_iflag = (IGNPAR | IGNBRK | ISTRIP | IXON | IXOFF);
tty_attr.c_oflag = 0;
tty_attr.c_cflag = (CREAD | B9600 | CS8 | CLOCAL );
tty_attr.c_lflag &= ~ICANON;
tty_attr.c_lflag &= ~ECHO;
tty_attr.c_cc[4] = 1;
tty_attr.c_cc[5] = 90;
if(ioctl(fd1, TCFLSH, 0) == -1) {
    printf ("Connect: ioctl flush failed, errno=%d\n",errno);
    return (-3);
}
if(ioctl(fd1, TCSETA, &tty_attr) == -1) {
     printf ("Connect: ioctl set failed, errno=%d\n",errno);
    return (-3);
}
if(ioctl(wfd, TCFLSH, 1) == -1) {
        printf ("Connect: ioctl flush failed, errno=%d\n",errno);
    return (-3);
}
```

```
        tty_attr.c_cc[4] = 0;

        tty_attr.c_cc[5] = 30;

        if(ioctl(wfd, TCSETA, &tty_attr) == -1) {

             printf ("Connect: ioctl set failed, errno=%d\n",errno);

            return (-3);

        }


        return(fd1);

}

/****************************************************************

 * It is called to receive maximum of 'size' bytes of data from the

 * DIP process. This process will be roadblocked until the requested

 * amount of data is provided by the DIP. The caller must pass a pointer

 * to the buffer where the requested data should be stored.

 *

 * RET:

 * n, where n >= 0. Returns the actual byte count of received data.

 * FAILED, if fails for some other reasons.

 *

 * IMPLEMENT:

 * - call read(),

 * - check return value.

 ****************************************************************/

receive(fd, data, size)

        char *data;                    /* address of recieve buffer */
        int size;                  /* size of buffer in bytes */
        int fd;                    /* file descriptor */
{
        int i;

        i = read(fd, data, size);
        if (i == -1) {
                printf ("Error reading: return = %d, errno = %d\n",i,errno);
        }

        fflush(stdout);
        return(i);
}
/* this routine does the actual writing of the data to the TTY port */
dsend(sfd, data, size)
        char *data;                  /* address of send buffer */
        int size;                  /* size of buffer in bytes */
        int sfd;                   /* file descriptor */
{
        int i;
        i = write(sfd, data, size);
        if (i < 0) {
               printf ("Error writing: return = %d, errno = %d\n",i,errno);
        }
        return(i);

}

/* alarm handling routine - increments counter to let other routines know that a
time-out has occurred */

alarm_rtn(sig_num)

        int sig_num;

{

        (void)signal(SIGALRM, alarm_rtn);

        alarm_cnt++;

        return (0);

}

setup()

{

        printf ("attempting connect\n");

        if ((fd = connect(dev_name)) < 0) {

                fflush(stdout);

                exit(-1);

        }

}

/*****************************************************************

*       This routine determines the type and validity of the data message received
on the TTY port. It then formats the appropriate response and writes it back out to
the TTY port.

******************************************************************/

get_resp()

{

        int     sflag,j,i,hlen,slen,retry;
        char    tmpstr[128],resp,rhead[20];
        char    *size_ptr,etx,rtn,rdata[128];


    etx = ETX;
        resp = NAK;
        /* Check 1st received character to decide whether to continue */
        switch(nbuf[0]){
            case ACK:
            case NAK:     return(1);     /* ignore */
                    break;;
            case SOH:     break;;
            case ETX:     dsend(wfd,&resp,1);/* send NAK */
                    return(-1);
                    break;;
            default:      return(-1);
                    break;;
        }
        printf("\nGot a SOH\n");
        alarm (alarm_val);           /* Start timer */
        alarm_cnt = 0;               /* Initialize indicator */
        sflag = 0;
        j = 1;
        /* Now get rest of header and all the data which follows */
        while ((alarm_cnt == 0) && ( sflag == 0)){     /* check for time-out */
          if ((i = receive(fd, &nbuf[j], 1)) < 0)
        continue;
 if(i > 0){
        switch(nbuf[j]){
                case ETX:       sflag = 1;
                                nbuf[j] = '\0';
                                break;;
                default:
                                break;;
        }
         j++;
        continue;
    }
}/* end of while loop */
if (alarm_cnt != 0){
        printf("\nTime out on rest of header\n");
        dsend(wfd,&resp,1);/* send NAK */
        return(-1);
}
 /* turn off and reset timer */
alarm(0);
alarm_cnt = 0;
 /* See if header was complete */
if(nbuf[15] != '\02'){
        printf("\nInvalid msg - no STX\n");
        dsend(wfd,&resp,1);/* send NAK */
        return(-1);

    }

    /* parse the header out of the data */

    strncpy(rhead,nbuf,16);

    rhead[16] = '\0';

    strcpy(rdata,&nbuf[16]);

    printf("\ngot header %s\n",rhead);

    printf("\ngot data %s\n",rdata);


    /* Send back ACKnowledgement that request was received */

    resp = ACK;

    dsend(wfd,&resp,1);

    /* Skip any request that appears to be a host response */

    if(rhead[7] == 'S')

        return(1);

    /* Give response (from host) the correct message type */

    rhead[7] = 'S';

    hlen = strlen(rhead);

    /*******************************************************

    *       The following code is what gives this stub its own functionality. It uses
the last digit of the functioncode in the header to determine what data should be
sent back to the DIP. One could just as easily use the whole function code, a
particular character or field in the data, or some combination thereof to
differentiate on the type of response to send.

    *******************************************************/

    switch(nbuf[10]){

        case '1':i =strlen(rdata);/* reverse it */
```

```
                    tmpstr[i] = '\0';
                    i--;
                    for(j=0;i >= 0; i--,j++)
                          tmpstr[j] = rdata[i];
                    break;
        case '2':i = atoi(rdata); /* double it */
                    i = i * 2;
                    sprintf(tmpstr,"%d",i);
                    break;
        default:return(1);
                    break;
 }
 sprintf(rdata,"%s%c",tmpstr,etx);
 printf("\nReturning data: %s\n",rdata);
 /* update header with actual size of the data */
slen = strlen(rdata);
size_ptr = &rhead[11];
sprintf(size_ptr,"%#.4d\02",slen - 1);  /*don't count ETX char*/


 /* attempt to send the data up to two times */
retry = 0;
while(retry < 2){
    /* send the header */
   if ((i = dsend(wfd, rhead, hlen )) < 0){
       sleep (5);
```

```
                 retry++;

                 continue;

             }

             /* send the data */

             if ((i = dsend(wfd, rdata, slen )) < 0){

                 sleep (5);

                 retry++;

                 continue;

             }

         alarm (alarm_val);          /* Start timer */

             alarm_cnt = 0;              /* Initialize indicator */


             /* get response from DIP */

             while (alarm_cnt == 0) {     /* check for time-out */

                 if ((i = receive(fd, &rtn, 1)) < 0) {

                     sleep (5);

                     continue;

                 }


                 if(i > 0){

                     switch(rtn){

                         case ACK:    printf("\nreceived ACK\n");

                                      retry = 3;

                                      break;;

                         case NAK:    printf("\nreceived NAK\n");
```

```
              retry++;
                              break;;
                  default:      break;;
                }
                break;
              }
            }/* end while  -- waiting for ACK/NAK loop */
            if (alarm_cnt != 0){
              printf("\nTime out on ack\n");
              retry++;
            }
            alarm(0);
            alarm_cnt = 0;
          }/* end while -- trying to send response loop */
    }
    /* signal catching clean up routine */
    cup(sig)
    int    sig;
    {
          close(fd);
          close(wfd);
          printf("\nReceived signal(%d) - exitting.\n",sig);
          fflush(stdout);
          exit(0);
    }
```

Stub Program

E

```
/*  This is a stub program that simulates a host for testing the send and receive
parts of the CSG asynchronous DIP.

 *

 *     This process works in partnership with hsend.c and hrec.c

 * MODULE NAME: hstub.c

 */

#include <stdio.h>

#include <sys/types.h>

#include <signal.h>

#include <ctype.h>

#include <errno.h>

#include <fcntl.h>

#include <termio.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <string.h>

#include "mesg.h"

#include "dip_et.h"
```

```
#include "asyncdip.h"
#include "hostreq.h"

extern int errno;

char  nbuf[MAXSIZE+2];              /* Buffer from async port */
int fd;                         /* File descriptor */
int wfd;                        /* File descriptor */
char dev_name[12];
struct termio tty_attr;             /* I/O line params */
struct termio save_attr;             /* I/O line params */

int alarm_val;                  /* Timer time_out value */
int alarm_cnt;                  /* Timer trap indicator */
int alarm_rtn();                /* Timer trap routine */

main(argc, argv)
int argc;
char *argv[];
{
     int i;              /* Temp counter */
     int cup();

     signal(SIGALRM, alarm_rtn);   /* Enable alarm */
     signal(SIGHUP,SIG_IGN);
signal(SIGINT,cup);
signal(SIGQUIT,cup);

if(argc != 2){
        printf("\nSYNTAX: hstub ttyXXX\n");
         fflush(stdout);
        exit(-1);
}
 sprintf(dev_name,"/dev/%s",argv[1]);

 alarm_val = ALARM_VAL;/* default time_out interval */
setup();
 printf("\nfd(read)=%d, fd(write)=%d\n", fd, wfd);
 fflush(stdout);

 /* loop forever, looking for data on TTY port */
while(1){
        if ((i = receive(fd, nbuf, 1)) < 0) {
                  fflush(stdout);
                exit(-1);
        }
         /* if error or no data, wait and then try again */
        if(i < 1) {
                  sleep (2);
                continue;
                }

                /* process incoming data */

                get_resp();

                fflush(stdout);

        }/* end main while loop */

}

/*****************************************************************

contains path name of line from the DIP to this simulated host TTY attributes to
change to and for saving original TTY atributes

 *****************************************************************/

connect(device)

char *device;

{

        int fd1;



        alarm (alarm_val);              /* Start timer */

        alarm_cnt = 0;                  /* Initialize indicator */



        if((fd1 = open(device, O_RDONLY)) < 0){

                printf ("Connect: open dev_name=<%s> failed, errno=%d\n",device,
errno);

            if (alarm_cnt == 0) {       /* check for time-out */

                return(-1);

            } else {

                printf ("Time-out on read open; possibly try later\n");

                return (-2);
```

```
          }
        }
        if((wfd = open(device, O_WRONLY | O_NDELAY)) < 0){
              printf ("Connect: open dev_name=<%s> failed, errno=%d\n",device,
errno);
          if (alarm_cnt == 0) {     /* check for time-out */
             return(-1);
          } else {
              printf ("Time-out on write open; possibly try later\n");
             return (-2);
          }
        }


        printf ("Open call succeeded, fd=%d\n",fd1);
        alarm (0);                  /* Kill timer */
        alarm_cnt = 0;                /* Reset indicator */
        /*
         * copy original attributes and setup raw mode
        */
        if(ioctl(fd1, TCGETA, &tty_attr) == -1) {
           printf ("Connect: ioctl get failed, errno=%d\n",errno);
          return (-3);
        }
        if(ioctl(fd1, TCGETA, &save_attr) == -1) {
           printf ("Connect: ioctl get failed, errno=%d\n",errno);
          return (-3);
```

```
}

tty_attr.c_iflag = (IGNPAR | IGNBRK | ISTRIP | IXON | IXOFF);

tty_attr.c_oflag = 0;

tty_attr.c_cflag = (CREAD | B9600 | CS8 | CLOCAL );

tty_attr.c_lflag &= ~ICANON;

tty_attr.c_lflag &= ~ECHO;

tty_attr.c_cc[4] = 1;

tty_attr.c_cc[5] = 90;

if(ioctl(fd1, TCFLSH, 0) == -1) {

   printf ("Connect: ioctl flush failed, errno=%d\n",errno);

   return (-3);

}

if(ioctl(fd1, TCSETA, &tty_attr) == -1) {

    printf ("Connect: ioctl set failed, errno=%d\n",errno);

   return (-3);

}

if(ioctl(wfd, TCFLSH, 1) == -1) {

       printf ("Connect: ioctl flush failed, errno=%d\n",errno);

   return (-3);

}

tty_attr.c_cc[4] = 0;

tty_attr.c_cc[5] = 30;

if(ioctl(wfd, TCSETA, &tty_attr) == -1) {

    printf ("Connect: ioctl set failed, errno=%d\n",errno);

   return (-3);
```

```
        }

        return(fd1);
}
/*****************************************************************
 * It is called to receive maximum of 'size' bytes of data from the
 * DIP process. This process will be roadblocked until the requested
 * amount of data is provided by the DIP. The caller must pass a pointer
 * to the buffer where the requested data should be stored.
 *
 * RET:
 * n, where n >= 0. Returns the actual byte count of received data.
 * FAILED, if fails for some other reasons.
 *
 * IMPLEMENT:
 * - call read(),
 * - check return value.
 *****************************************************************/
receive(fd, data, size)
        char *data;             /* address of recieve buffer */
        int size;              /* size of buffer in bytes */
        int fd;                /* file descriptor */
{
        int i;
        i = read(fd, data, size);

        if (i == -1) {

                printf ("Error reading: return = %d, errno = %d\n",i,errno);

        }


        fflush(stdout);

        return(i);

}

/* this routine does the actual writing of the data to the TTY port */

dsend(sfd, data, size)

        char *data;                 /* address of send buffer */

        int size;                   /* size of buffer in bytes */

        int sfd;                    /* file descriptor */

{

        int i;

        i = write(sfd, data, size);

        if (i < 0) {

                printf ("Error writing: return = %d, errno = %d\n",i,errno);

        }


        return(i);

}

/* alarm handling routine - increments counter to let other routines know that a
time-out has occurred */

alarm_rtn(sig_num)

        int sig_num;
```

```
{

        (void)signal(SIGALRM, alarm_rtn);

        alarm_cnt++;

        return (0);

}

setup()

{

        printf ("attempting connect\n");

        if ((fd = connect(dev_name)) < 0) {

                 fflush(stdout);

                exit(-1);

        }

}

/*****************************************************************

*       This routine determines the type and validity of the data message received
on the TTY port. It then formats the appropriate response and writes it back out to
the TTY port.

*****************************************************************/

get_resp()

{

        int     sflag,j,i,hlen,slen,retry;

        char    tmpstr[128],resp,rhead[20];

        char    *size_ptr,etx,rtn,rdata[128];


etx = ETX;

        resp = NAK;
```

```
 /* Check 1st received character to decide whether to continue */
switch(nbuf[0]){
        case ACK:
        case NAK:       return(1);      /* ignore */
                        break;;
        case SOH:       break;;
        case ETX:       dsend(wfd,&resp,1);/* send NAK */
                        return(-1);
                        break;;
        default:        return(-1);
                        break;;
}
 printf("\nGot a SOH\n");
alarm (alarm_val);              /* Start timer */
alarm_cnt = 0;                  /* Initialize indicator */
sflag = 0;
j = 1;
 /* Now get rest of header and all the data which follows */
while ((alarm_cnt == 0) && ( sflag == 0)){     /* check for time-out */
    if ((i = receive(fd, &nbuf[j], 1)) < 0)
        continue;
 if(i > 0){
        switch(nbuf[j]){
                case ETX:       sflag = 1;
                                nbuf[j] = '\0';
                        break;;
                default:
                        break;;
            }
             j++;
            continue;
        }
    }/* end of while loop */
    if (alarm_cnt != 0){
            printf("\nTime out on rest of header\n");
            dsend(wfd,&resp,1);/* send NAK */
            return(-1);
    }
     /* turn off and reset timer */
    alarm(0);
    alarm_cnt = 0;
     /* See if header was complete */
    if(nbuf[15] != '\02'){
            printf("\nInvalid msg - no STX\n");
            dsend(wfd,&resp,1);/* send NAK */
            return(-1);
    }
     /* parse the header out of the data */
    strncpy(rhead,nbuf,16);
     rhead[16] = '\0';
```

```
strcpy(rdata,&nbuf[16]);

printf("\ngot header %s\n",rhead);

printf("\ngot data %s\n",rdata);


/* Send back ACKnowledgement that request was received */

resp = ACK;

dsend(wfd,&resp,1);

/* Skip any request that appears to be a host response */

if(rhead[7] == 'S')

        return(1);

/* Give response (from host) the correct message type */

rhead[7] = 'S';

hlen = strlen(rhead);

/*******************************************************

*       The following code is what gives this stub its own functionality. It uses
the last digit of the functioncode in the header to determine what data should be
sent back to the DIP. One could just as easily use the whole function code, a
particular character or field in the data, or some combination thereof to
differentiate on the type of response to send.

*******************************************************/

switch(nbuf[10]){

        case '1':i =strlen(rdata);/* reverse it */

                        tmpstr[i] = '\0';

                        i--;

                        for(j=0;i >= 0; i--,j++)

                                tmpstr[j] = rdata[i];

                        break;
        case '2':i = atoi(rdata); /* double it */
                        i = i * 2;
                        sprintf(tmpstr,"%d",i);
                        break;
        default:return(1);
                        break;
}
sprintf(rdata,"%s%c",tmpstr,etx);
printf("\nReturning data: %s\n",rdata);
/* update header with actual size of the data */
slen = strlen(rdata);
size_ptr = &rhead[11];
sprintf(size_ptr,"%#.4d\02",slen - 1);  /*don't count ETX char*/


/* attempt to send the data up to two times */
retry = 0;
while(retry < 2){
    /* send the header */
    if ((i = dsend(wfd, rhead, hlen )) < 0){
        sleep (5);
        retry++;
        continue;
    }
    /* send the data */
    if ((i = dsend(wfd, rdata, slen )) < 0){
```

```
                sleep (5);
                 retry++;
                 continue;
              }
           alarm (alarm_val);              /* Start timer */
              alarm_cnt = 0;                  /* Initialize indicator */

              /* get response from DIP */
              while (alarm_cnt == 0) {       /* check for time-out */
                 if ((i = receive(fd, &rtn, 1)) < 0) {
                    sleep (5);
                    continue;
                 }

                 if(i > 0){
                    switch(rtn){
                       case ACK:     printf("\nreceived ACK\n");
                                   retry = 3;
                                   break;;
                       case NAK:     printf("\nreceived NAK\n");
                 retry++;
                                   break;;
                       default:     break;;
                    }
                    break;
```

```
            }
        }/* end while  -- waiting for ACK/NAK loop */
        if (alarm_cnt != 0){
            printf("\nTime out on ack\n");
            retry++;
        }
        alarm(0);
        alarm_cnt = 0;
    }/* end while -- trying to send response loop */
}
/* signal catching clean up routine */
cup(sig)
int     sig;
{
    close(fd);
    close(wfd);
    printf("\nReceived signal(%d) - exitting.\n",sig);
    fflush(stdout);
    exit(0);
}
```