Technical Reference: Hayes® Synchronous Interface for Applications Software

Release 1.0

March 12, 1987

Hayes Microcomputer Products, Inc.
705 Westech Drive
Norcross, Georgia 30092

Please register for updates
You are encouraged to register with Hayes Microcomputer Products, Inc. All licensed developers will receive updates to this specification when additions or modifications are made.

For more information, call Hayes Customer Service at 404/441-1617 (United States); 416/283-2627 (Canada); 852-5-845-9818 (Hong Kong); or 01-848-1868 (United Kingdom) or by calling Hayes Bulletin Board System at 1-800-USHAYES or 404/HIMODEM (United States) or 01-569-1774 (United Kingdom).

Document 44-00005 AA A40

Hayes®

TECHNICAL REFERENCE--HAYES SYNCHRONOUS INTERFACE FOR APPLICATION SOFTWARE

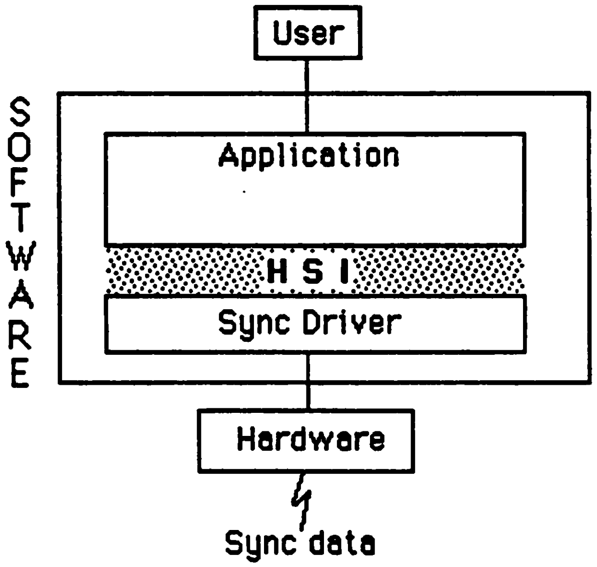
TABLE OF CONTENTS

INTRODUCTION

What is the Hayes Synchronous Interface?

The Hayes Synchronous Interface (HSI) is a specification describing the interface between two layers of software in a synchronous communications package. One layer of software, the "synchronous driver," consists of the routines necessary to operate the synchronous communications hardware. The synchronous driver services device interrupts and transfers data to and from the hardware using the computer's I/O instructions.

The other layer of software, the "application," transmits and receives messages. HSI describes how these messages, and control information, are exchanged between the application and the synchronous driver.

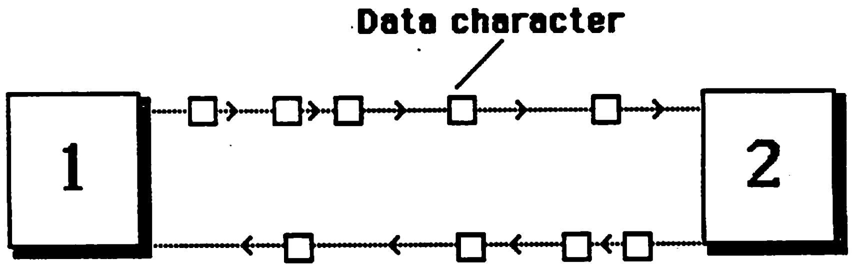


What is the Hayes Synchronous Driver?

Supporting HSI is the Hayes Synchronous Driver (HSD). HSD is software that performs the interrupt servicing and other detailed I/O operations necessary to operate specific hardware, such as the Hayes Smartmodem 2400B, in its synchronous mode. Its interactions with the main body of the synchronous application software conform to the HSI design specification described in this document. The Hayes Synchronous Driver therefore bridges the gap between the synchronous application software and the Smartmodem 2400B hardware. The Hayes Synchronous Driver can be licensed from Hayes Microcomputer Products, Inc., on an annual fee basis by software developers.

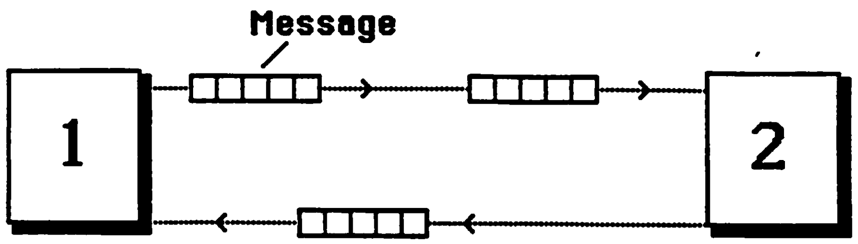
Review of Synchronous Concepts and Terminology

Synchronous communications differ from asynchronous communications in that data is organized into messages. In asynchronous communications, data is exchanged character-by-character. There is no strict timing relationship between the transmission of one character and the next. Start and stop bits must be transmitted to announce when each character is transmitted.



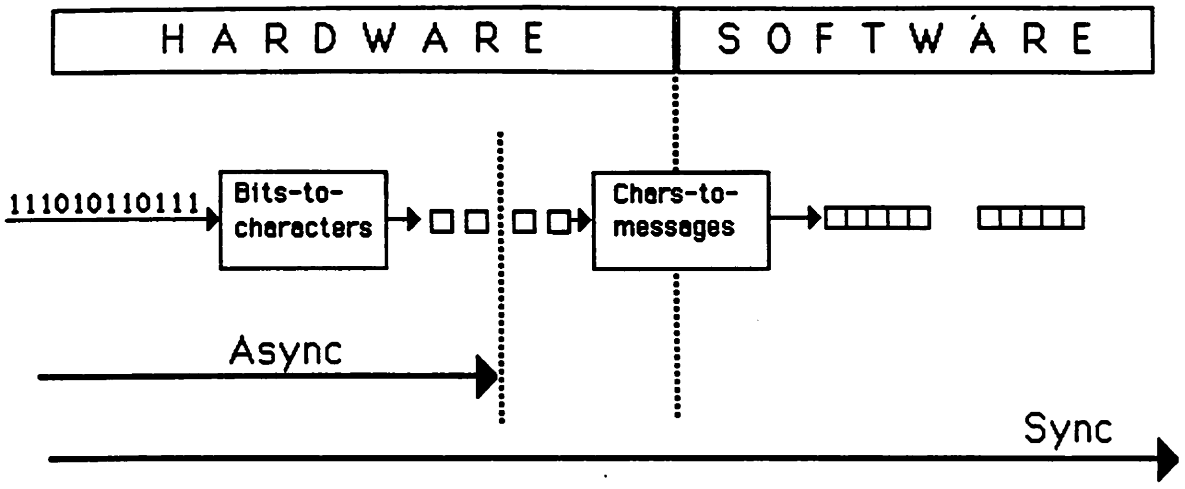
Asynchronous data

In synchronous communications, however, data is exchanged in messages. The characters which comprise each message are transmitted back-to-back, with no start or stop bits. Removing this start bit/stop bit overhead makes synchronous communication more efficient than asynchronous communication. Furthermore, the message-oriented nature of synchronous communication can be used to greater advantage by sophisticated applications (such as office automation) than the more primitive character-by-character asynchronous communication.

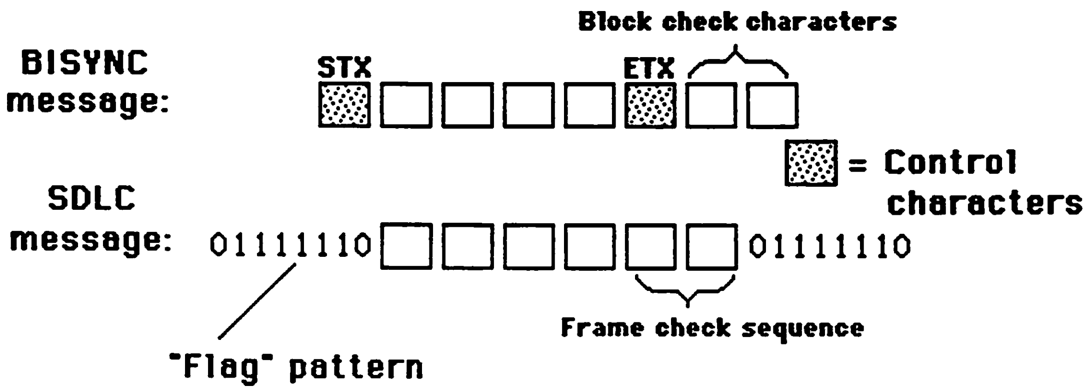


Synchronous data

In both synchronous and asynchronous communication, the stream of ones and zeros coming in from the communications line must be converted into a stream of characters. Various techniques are used to establish the boundaries between the characters in the stream of bits. This conversion is almost always carried out by specialized hardware in the synchronous or asynchronous communications adapter. In addition, synchronous communications convert the stream of characters into messages, and various techniques are used to indicate the beginning and end of each message.



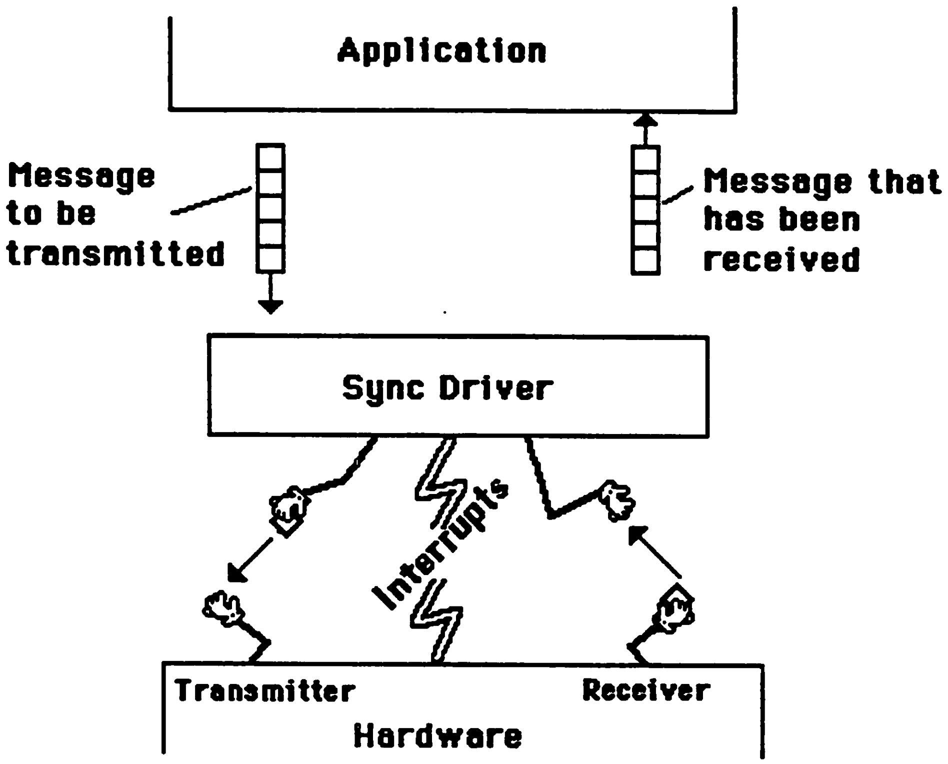
In general, the beginning and end of each synchronous message is indicated by protocols which place a special bit pattern or control character at the beginning and at the end of the message. Synchronous protocols using control characters are byte-oriented; those using bit patterns (not necessarily corresponding to characters) are bit-oriented. Byte-oriented protocols, such as IBM's Binary Synchronous Communications (BISYNC), are generally older and being replaced by the more modern bit-oriented protocols, such as IBM's Synchronous Data Link Control (SDLC) or X.25's High-Level Data Link Control (HDLC).



Role of the Synchronous Driver

The characters-to-messages conversion is carried out jointly between the communications interface adapter hardware and the synchronous driver software. The responsibility of the synchronous driver software is to:

1. Respond to interrupts generated by the adapter using I/O instructions.
2. Assist the communications interface adapter hardware in characters-to-messages conversion.
3. Accept messages to be transmitted from the application and send it messages that have been received.



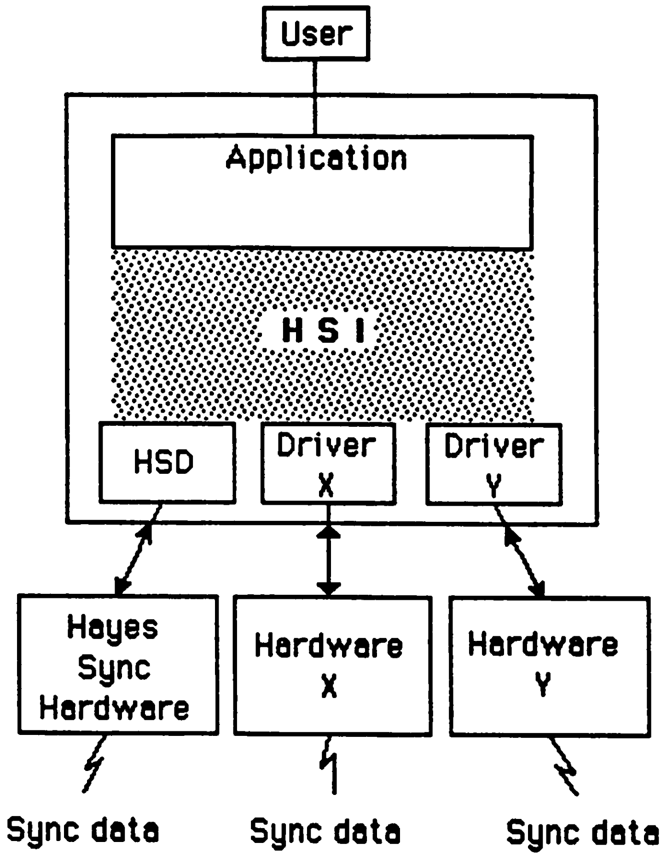
Recap: What are HSI and HSD?

This document describes the Hayes Synchronous Interface (HSI), a standard way of fitting the application and synchronous driver together. HSI is the interface used by the Hayes Synchronous Driver (HSD), a complete driver module that supports Hayes synchronous devices. The source code for HSD is available for license from Hayes Microcomputer Products, Inc., Customer Service group. By following the specifications in this document, software developers can write synchronous drivers for non-Hayes synchronous devices that support the same application interface as HSD.

Advantages of Using HSI and HSD

By using HSI in a synchronous communication software package, compatibility with current and future synchronous products from Hayes can be achieved. Hayes has made available an HSI-compatible synchronous driver supporting current products and will update this driver to support future synchronous products.

General compatibility with other synchronous hardware devices can be achieved by writing or obtaining HSI-compatible synchronous drivers. Since these synchronous drivers would share the same application interface, support is added to the synchronous communication package without changing the application side of the standard interface.

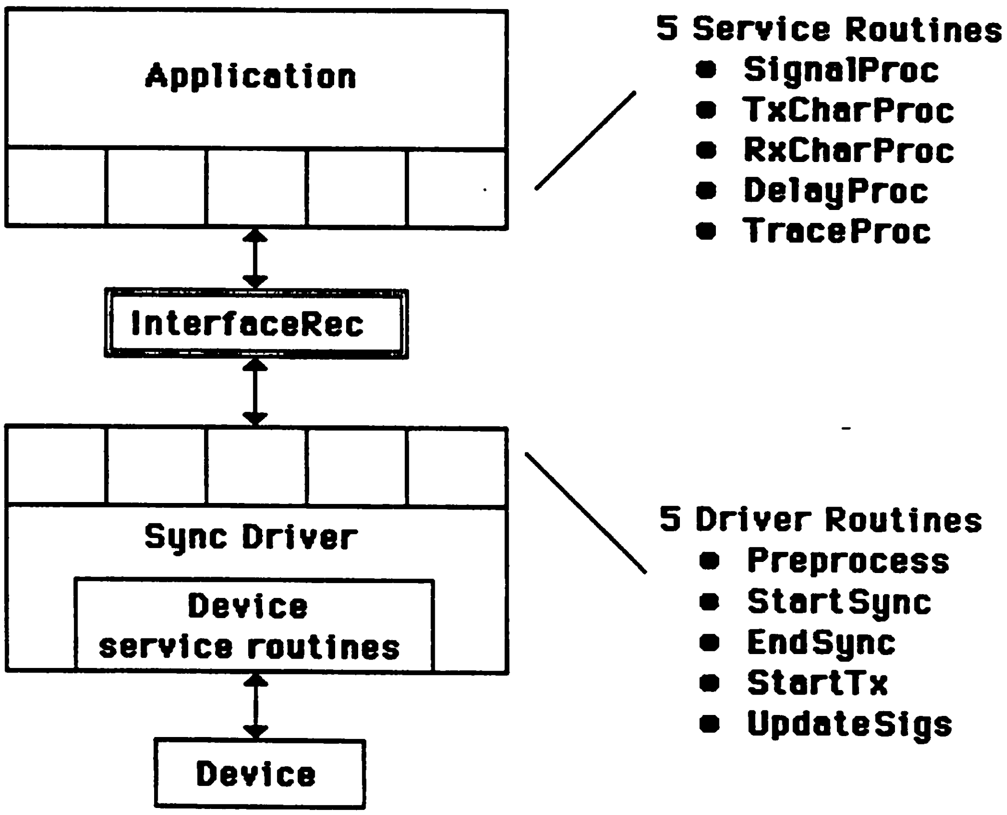


OVERVIEW

HSI consists of three main components:

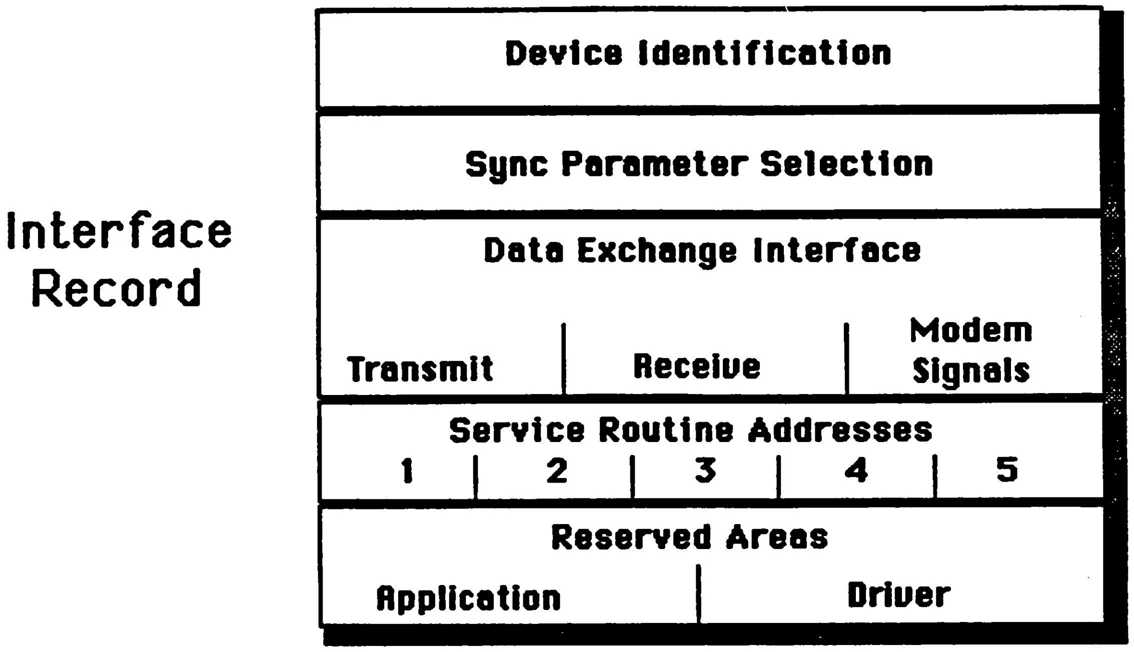
1. Interface Record, or InterfaceRec, a control block shared between the driver and the application
2. Driver Routines, five routines inside the driver that the application calls
3. Service Routines, five routines inside the application that the driver calls

Physically, a synchronous driver (such as HSD) consists of a code segment containing the five driver routines, plus a device service routine for each device supported by the driver.



The InterfaceRec

The InterfaceRec is a control block containing information used to control the operation of one synchronous device. If more than one device is being operated simultaneously, there would be an InterfaceRec for each one, but each InterfaceRec would have the same format. The address of the InterfaceRec is passed as a parameter every time a driver routine or service routine is called. It is the application's responsibility to allocate memory for the InterfaceRec.



Inside the InterfaceRec

A detailed breakdown of the InterfaceRec is presented in Appendix A. Some fields in the InterfaceRec are controlled by the application, the synchronous driver, or both. The fields in the InterfaceRec can be divided into five sections:

1. Device Identification

   These fields are set by the application, read by the synchronous driver, and used during set-up to help the driver identify the device it will operate.

2. Synchronous Parameter Selection

   These fields are also set by the application and read by the synchronous driver. They are also used during the set-up phase to establish synchronous protocol options.

3. Data Exchange Interface

   This section contains fields that are controlled by the synchronous driver, the application, or both. These fields are used during active,"on-line" operation to coordinate the flow of data messages sent back and forth between the driver and the application; to control and report the status of the modem signal lines; and to report and record error conditions. The data exchange interface can be divided into three subsections: receive side, transmit side, and modem signals.

4. Service Routine Addresses

   Five fields are defined in this section, in which the application stores the addresses of the five service routines. This is the linkage mechanism by which the synchronous driver learns the addresses of the service routines, which are essentially subroutines inside the application that the driver may call.

5. Reserved Areas

   Two 48-byte reserved areas are included in the InterfaceRec: one for the synchronous driver's internal use and one for the application to use however it wishes.

The Driver Routines

The five driver routines, called by the application, are as follows:

1.	Preprocess	Called before starting synchronous operation, Preprocess identifies the device to be operated and completes driver initialization.
2.	StartSync	Starts synchronous operation.
3.	EndSync	Terminates synchronous operation.

4.	StartTx	The application calls StartTx to inform the synchronous driver that it has a message to transmit.
5.	UpdateSigs	Calling UpdateSigs causes the synchronous driver to update the status of the modem signals.

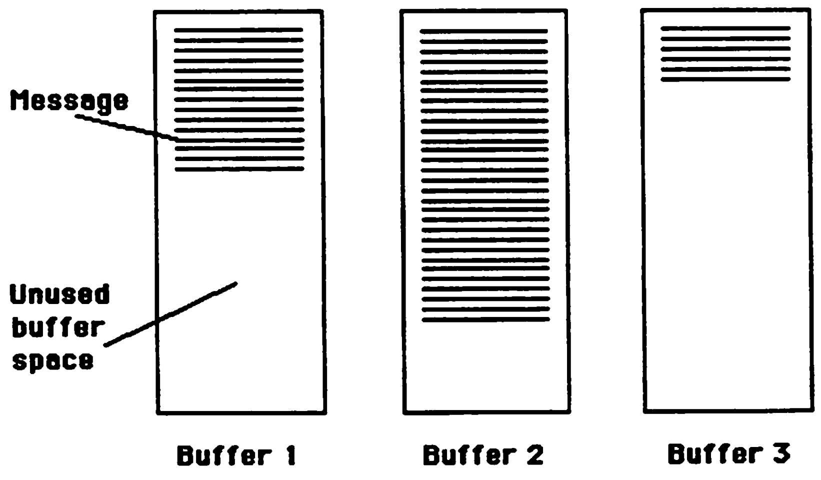
The Service Routines

The five service routines, called by the driver, are:

1.	SignalProc	Part of the signaling mechanism the synchronous driver uses to inform the application when certain events occur.
2.	TxCharProc	Used to customize transmitter operation in BISYNC mode.
3.	RxCharProc	Used to customize receiver operation in BISYNC mode.
4.	DelayProc	Used when a delay is required during termination.
5.	TraceProc	Part of a simple trace facility useful for debugging driver code.

Buffer Usage

In essence, the synchronous driver transmits and receives messages via specific hardware and passes these messages to and from the application using a standard interface, HSI. Buffers are used to store these messages; a buffer is a reusable block of memory large enough to contain an entire message. Usually, one standard buffer size is used in a given application. HSI is compatible with any buffer size, up to 64K bytes. Each message must be stored in a separate buffer and must fit into a single buffer.



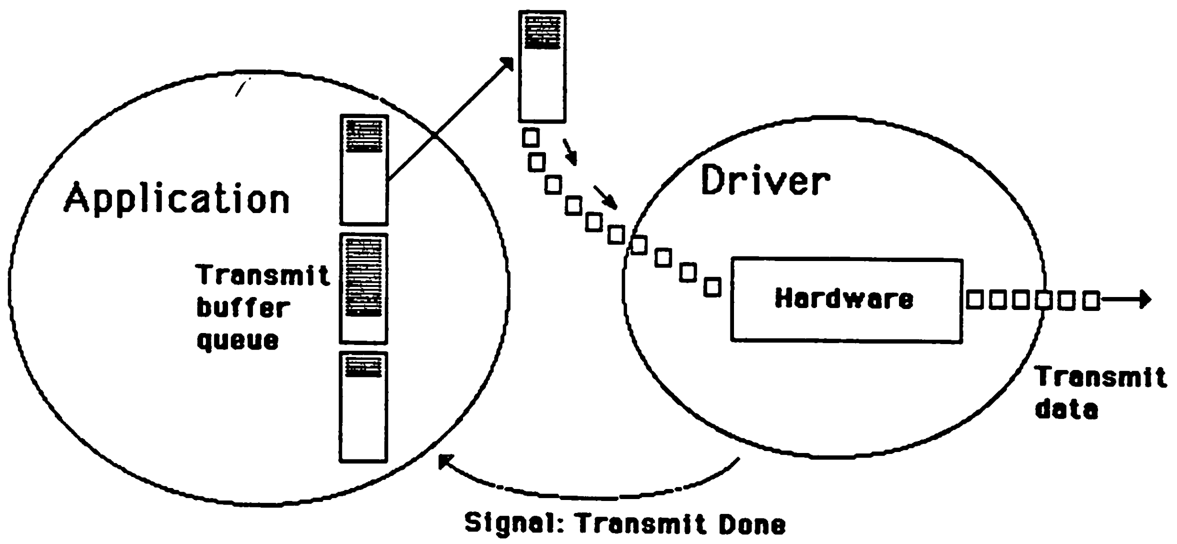
HSI supports two basic synchronous protocols: SDLC/HDLC and BISYNC. In the case of SDLC/HDLC, messages are called frames, so each buffer corresponds to one SDLC/HDLC frame. In the case of BISYNC, messages are called blocks, so each buffer corresponds to one BISYNC block.

What the Synchronous Driver Does

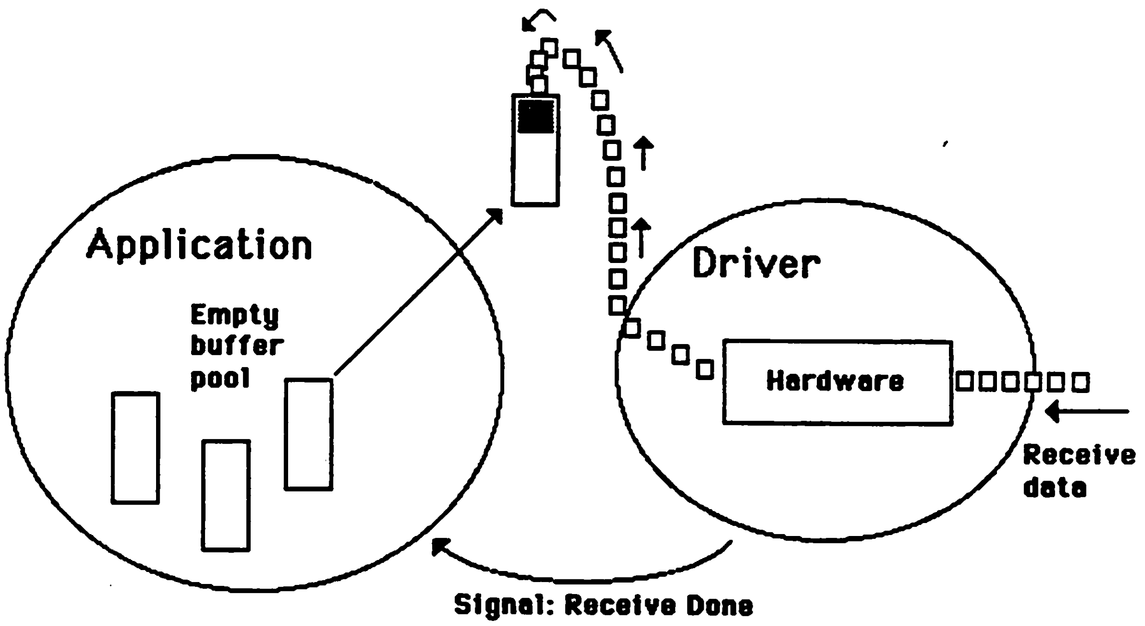
The synchronous driver performs all interrupt handling and device I/O necessary to transmit and receive messages: on the transmit side, it takes data characters from the buffers and feeds them to the hardware transmitter; on the receive side, it accepts data characters from the hardware receiver and stores them in empty buffers.

What the Application Does

The application is responsible for buffer allocation and management. In general, the application does everything that might involve interacting with the host operating system, such as memory allocation and deallocation, and process synchronization. When it has a message to transmit, the application stores the message in a buffer and passes the buffer to the synchronous driver. The application also maintains a queue of messages to be transmitted, if this is desired. The synchronous driver signals the application when it has completed message transmission; the application may then remove the spent buffer and replace it with a buffer containing the next message to transmit.



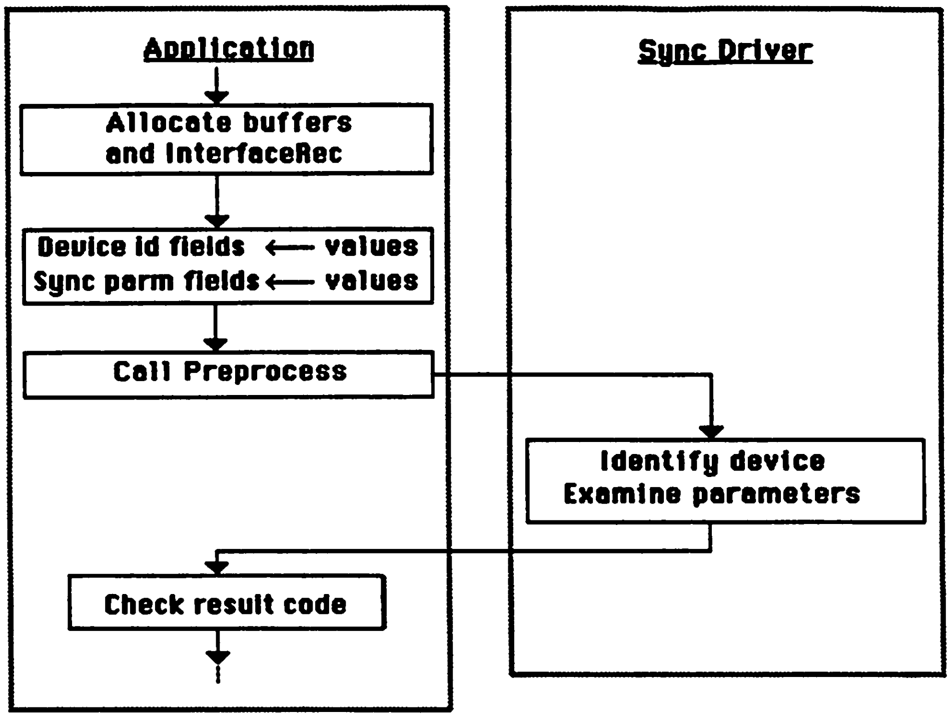
Similarly for the receive side, the application supplies the synchronous driver with an empty buffer to receive the next incoming message. When message reception is completed, the driver signals the application, which queues the message for processing and replaces the full buffer with an empty one.



Starting Synchronous Operation

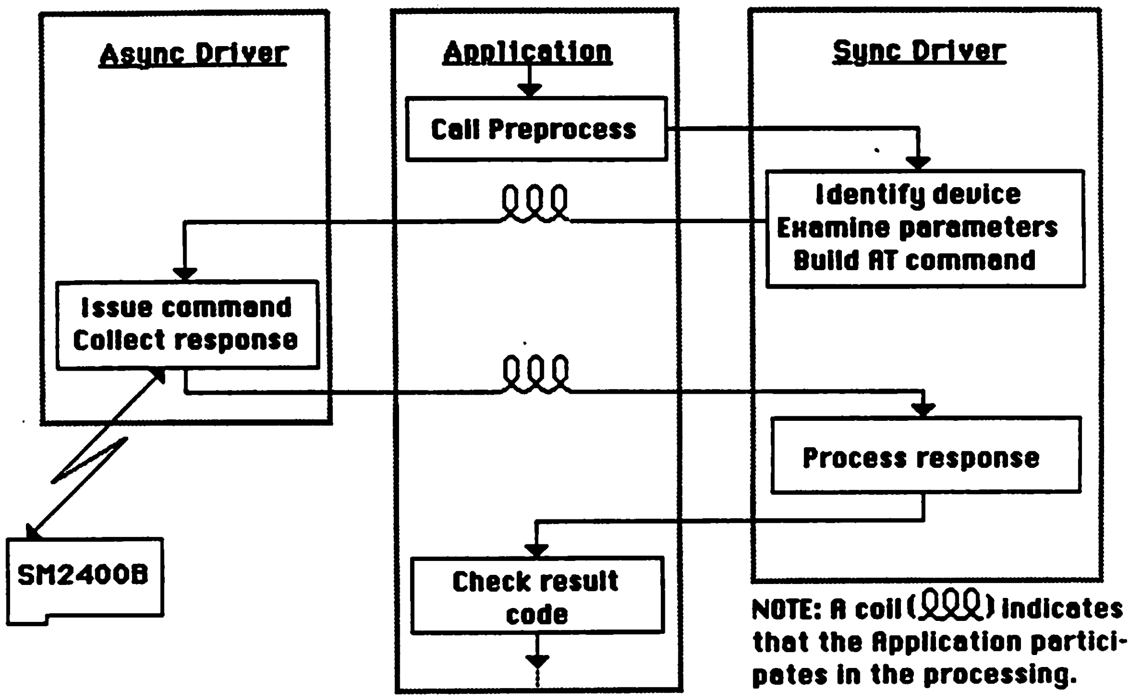
Calling Preprocess

An application using HSI first initializes itself, allocating memory for buffers and the InterfaceRec. The application fills in a number of InterfaceRec fields, including those for device identification and synchronous parameter selection, then calls Preprocess. Preprocess identifies the hardware configuration and examines the selected synchronous protocol options. It returns a result code to the application indicating whether it completed its operations normally or whether an error occurred (perhaps a protocol option was selected that the hardware being used didn't support).



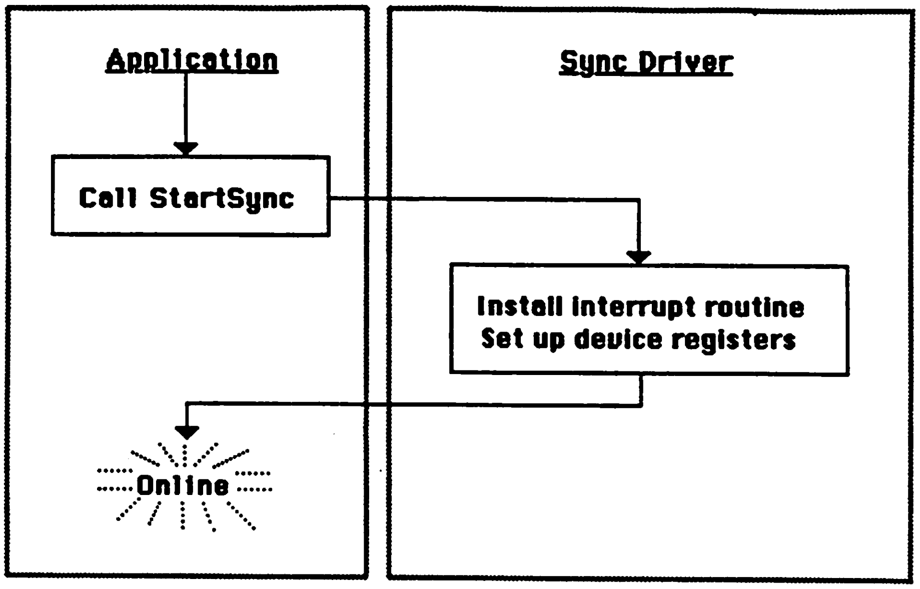
Sometimes synchronous devices have an alternate communications channel or method used to exchange control and status information. Preprocess may need to interact with the device over that alternate channel to carry out its functions, (e.g., the asynchronous command state of the Hayes Smartmodem 2400B).

In such cases it may be desirable to use a separate driver to operate the alternate channel or communication method (e.g., an asynchronous driver for the Smartmodem 2400B). HSI allows for this by defining a mechanism by which Preprocess can interact with the separate driver. For example, when Preprocess passes an AT command to the asynchronous driver, the asynchronous driver issues the command, collects the modem's response string, and passes it back to Preprocess.



Calling StartSync

When Preprocess returns with an "OK" result code, both the driver and the application are completely initialized and ready to go. To actually start synchronous operation, the application calls StartSync. The driver takes control of the device, installs its interrupt service routine, initializes the device registers according to the selected synchronous protocol options, and begins transferring data.

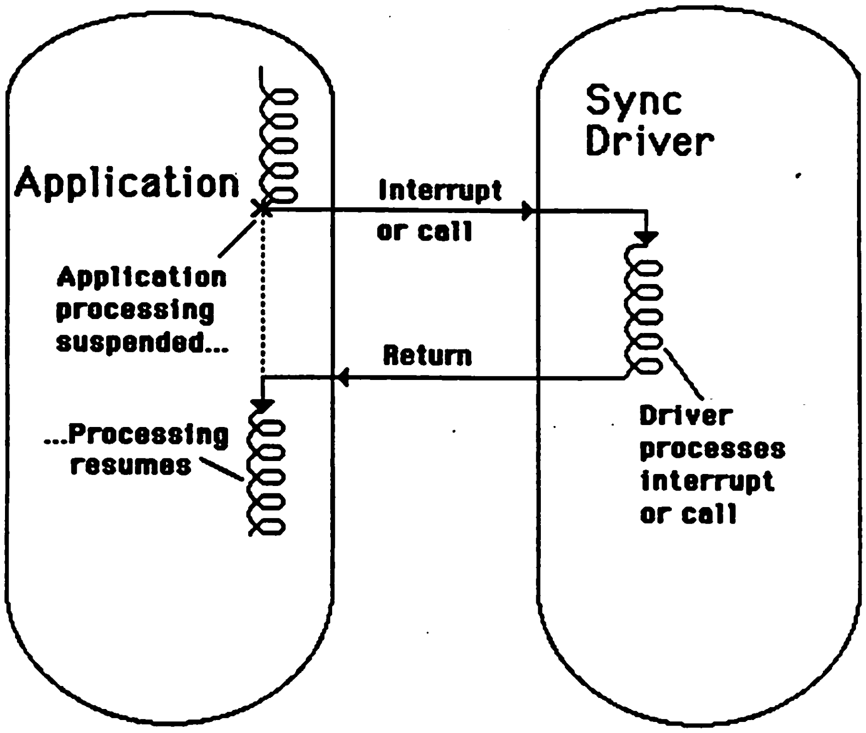


How the Driver Gets Control of the CPU

After control returns from the StartSync call, the driver gets control of the processor again to perform its functions in one of two ways:

- in response to device interrupts, or
- whenever the application calls StartTx to start a transmission or calls UpdateSigs to update the modem signals.

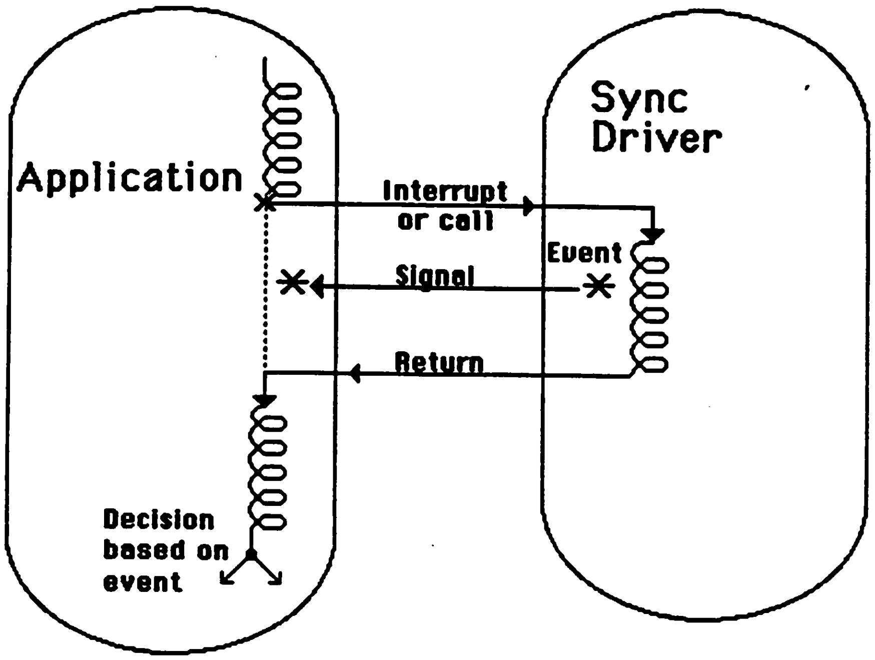
The driver keeps control as long as it takes to carry out the necessary operations to service the interrupt, start the transmission, or update the modem signals. Control is then returned to the interrupted process or the caller of the driver procedure.


Application
Sync Driver
Interrupt or call
Application processing suspended...
Return
...Processing resumes
Driver processes interrupt or call

The Signaling Mechanism

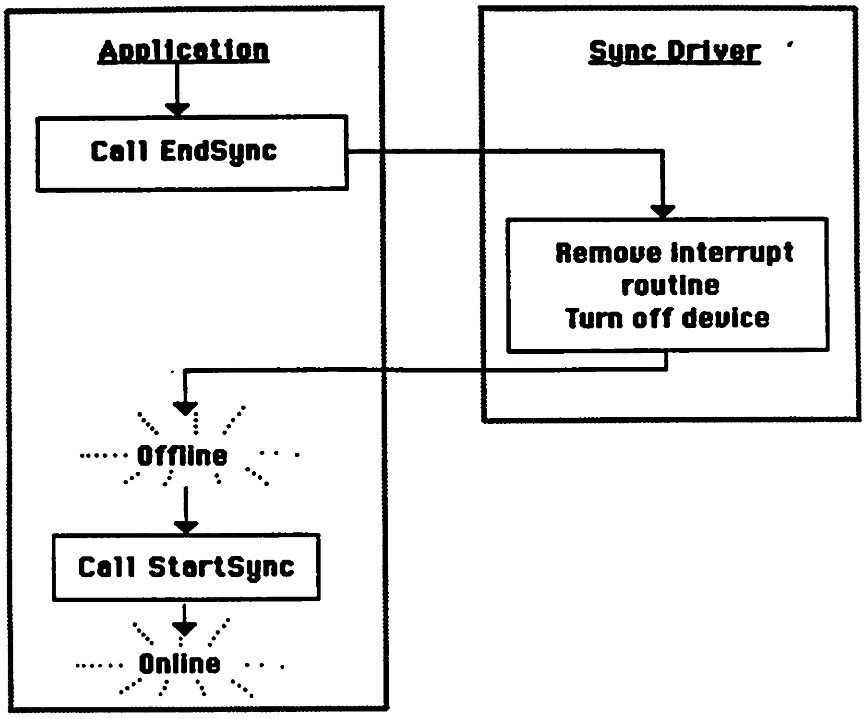
On a number of occasions the driver must interact with the application. For example, message reception may have just been completed or a modem signal might have changed state. To manage such events, the application must execute specialized code. The signaling mechanism enables that code to be executed at the right time.

When the driver detects one or more events, it constructs a special signal word indicating which events have occurred. Before it returns control to the interrupted process or the caller of the driver procedure, it calls SignalProc, and passes the signal word as a parameter. Inside SignalProc, the application handles each event by executing the appropriate code or causing the multitasking system (if one is being used) to schedule a high-priority task to handle the event.



Stopping Synchronous Operation

When the application is ready to terminate synchronous operation, it calls EndSync, causing the driver to turn off the device, remove the interrupt service routine, and terminate synchronous operation. The synchronous communication session may be restarted by calling StartSync again. Preprocess should not be called again unless a new synchronous session (with different parameters) is desired.



Stopping and restarting synchronous operation is necessary if the application needs to use the device (e.g., a Hayes modem) to perform a short, transient operation, such as issuing a status-inquiry AT command.

Error Handling

The synchronous driver is responsible for handling all error situations that might arise during device operation. An error code field is defined in the InterfaceRec for both transmit and receive. When a transmit or receive operation is completed (either normally or with an error), an appropriate code is stored in this field to indicate the outcome of the operation. In addition, an array of error counters, one for each error code for both transmit and receive, is defined in the InterfaceRec. The driver automatically maintains these counters, which indicate how many transmit and receive errors have occurred.

BISYNC vs. SDLC/HDLC

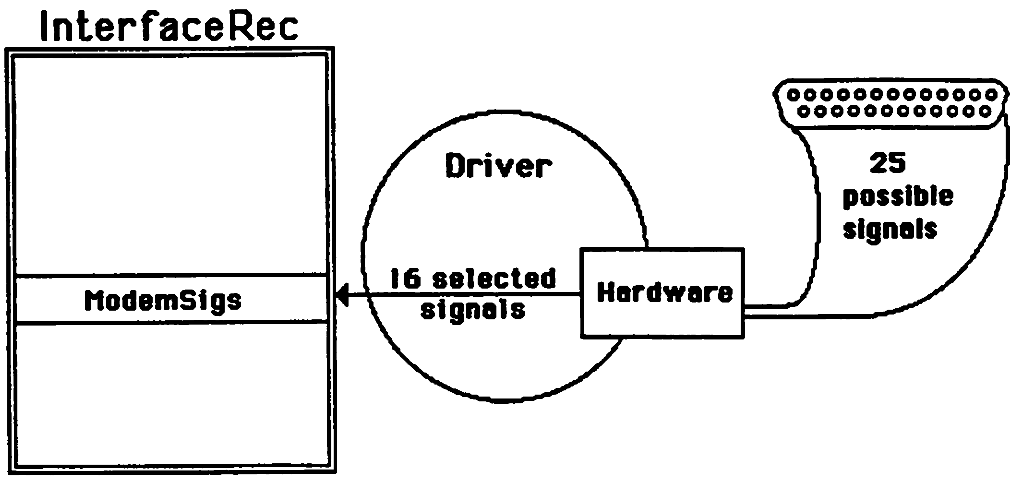
HSI supports both BISYNC and SDLC/HDLC protocols. Support of SDLC/HDLC is straightforward, as these protocols (essentially identical for our purposes) enjoy well-established standards for frame formats, transmission and reception rules, etc. Many BISYNC applications, however, use slightly different variations of the basic protocol. Thus, the application must customize the operation of the synchronous driver in BISYNC mode according to the BISYNC protocol variant being used.

This is accomplished through two per-character routines, RxCharProc and TxCharProc. On the receive side, the driver calls RxCharProc after each character is received. Inside RxCharProc, the application examines the characters as they are received, and decides to accept them, delete them, or initiate end-of-message processing. On the transmit side, the driver calls TxCharProc just before transmitting each character. Inside TxCharProc, the application decides to issue the character for transmission, substitute a different character, or insert a new character into the message. In general, the application must do more work when BISYNC mode is selected instead of SDLC/HDLC.

Modem Signal Support

Modem signals are binary, electronic signals carried between the synchronous hardware and the device it is connected to, usually a modem. The use of these signals is standardized by the RS-232 specification, but the standard is not adhered to precisely by every device. To accommodate nonstandard use of these signals (such as occurs in IBM's modems to support Link Problem Determination Aid, LPDA), HSI makes few assumptions about the modem signals.

Sixteen of the 25 possible modem signals are provided for in a modem signal word found in the InterfaceRec, which shows the current status of these signal lines (data, clock, and ground signals are omitted). The application can set and test these signals individually. The InterfaceRec also contains a word set by the synchronous driver, indicating which of the 16 signals are actually functional in the device being used.



HSI specifies the operation of three modem signals: RTS, CTS, and DCD. The synchronous driver's transmit routines automatically raise RTS (Request To Send) before each transmission, if the application has not programmed this signal high. In addition, these routines wait for CTS (Clear To Send) before beginning the transmission. An error is generated if CTS is lost during transmission. Similarly, an error is generated by the receive routines if DCD (Data Carrier Detect) is not present during a receive operation.

Trace Facility

HSI provides a simple trace facility, useful for debugging driver code. The TraceProc service routine can be used to pass trace information from the synchronous driver to the application. The application makes sure that the trace information is displayed or printed for the programmer to study.

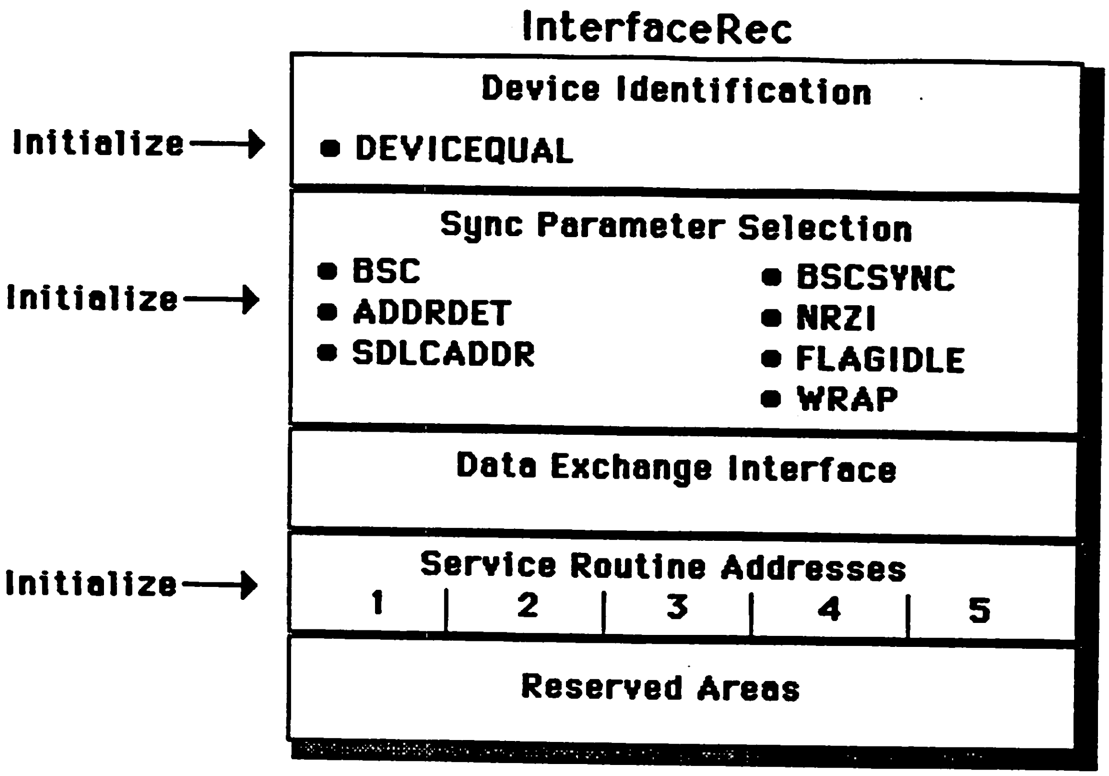
CHAPTER 1--STARTING SYNCHRONOUS OPERATION

This chapter describes the process of starting synchronous operation using HSI. In chronological order, it describes the operations the application and the synchronous driver must perform.

Creating the InterfaceRec

The application is responsible for allocating memory for the InterfaceRec. Using the memory management services available through the host operating system and/or programming language, a block of memory of the correct size is allocated. The size of the InterfaceRec is indicated in Appendix A. After allocating the InterfaceRec, the application clears the entire block of memory to zeros.

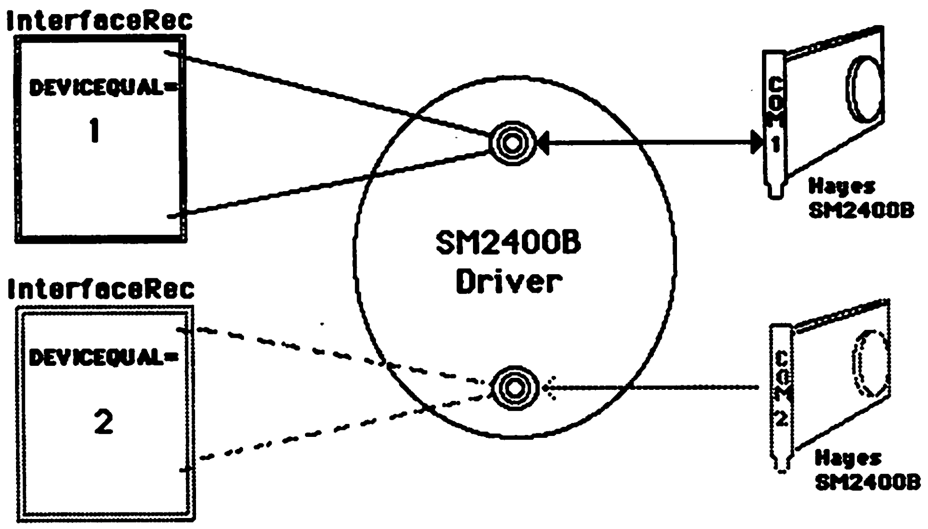
Next, the application fills in the device qualifier field, DEVQUAL, and the synchronous parameter selection fields. The synchronous parameter selection fields are discussed in Chapters 2 and 3 on SDLC/HDLC and BISYNC operation, respectively. The device qualifier field is discussed below. The application also stores the addresses of the five service routines in the InterfaceRec fields defined for this purpose.



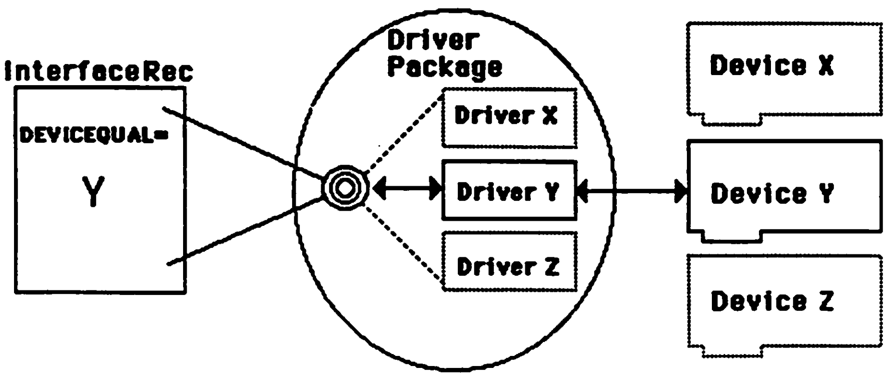
Device Qualifiers

The synchronous driver determines the hardware configuration present, but may require assistance to do so. To assist the synchronous driver establish device identity, the DEVQUAL field is provided in the InterfaceRec. This 16-bit field is not defined in detail by HSI, since the information needed by a synchronous driver to establish device identity cannot be anticipated.

To clarify the DEVQUAL concept, consider a synchronous driver for a device such as the Hayes Smartmodem 2400B. This device can be set to one of two possible configurations: COM1 or COM2. It is impossible for the driver to know which COM port it must use to access the Smartmodem 2400B; therefore the device qualifier field is used to provide this information. DEVQUAL = 1 could be used to indicate COM1, and DEVQUAL = 2 to indicate COM2.



When the drivers for several different synchronous devices are packaged in one large synchronous driver module (such as HSD), it is not possible through hardware alone to determine which synchronous device is present at any given time. Therefore, the device qualifier field may be used to indicate which device-specific driver will be activated within the large synchronous driver module.



Appendix C lists a number of reserved device qualifier values, including those used with HSD. As a service to HSD licensees, Hayes is coordinating the assignment of additional DEVQUAL values for use with synchronous drivers other than HSD. These values will be openly published with the values currently listed in Appendix C.

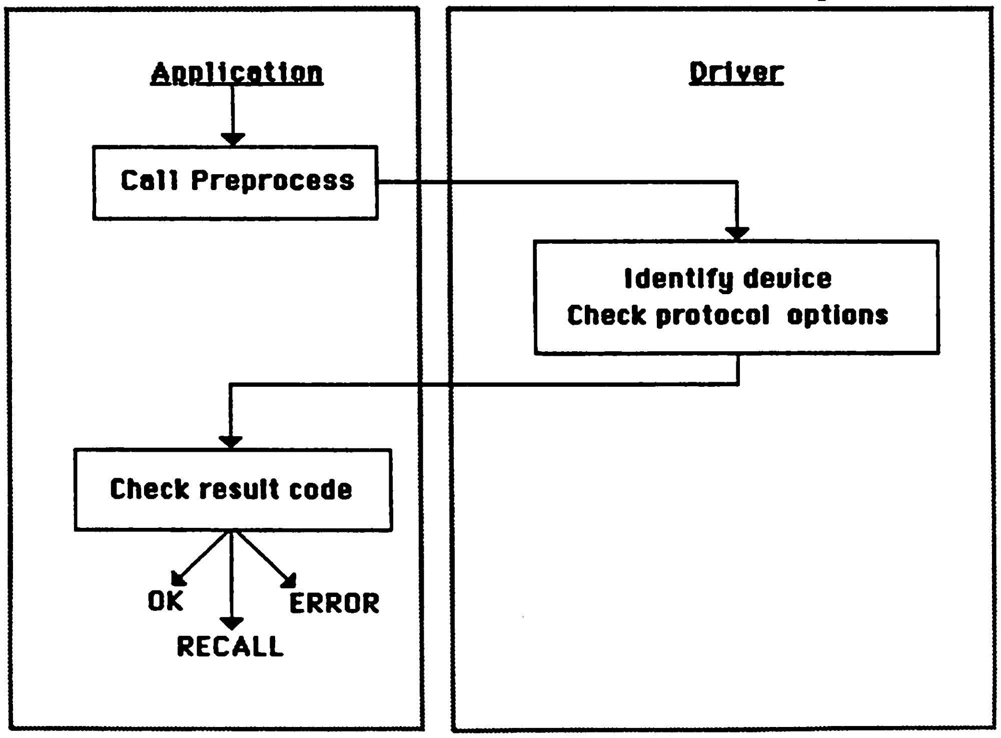
I/O Port and Interrupt Vector

For some choices of DEVQUAL, the InterfaceRec fields IOPORT and INTVEC must be filled in with the I/O port address and interrupt vector number of the hardware in use. This step is necessary when these quantities can vary over a relatively wide range, according to the position of DIP switches or jumpers on the hardware device.

The synchronous driver determines whether or not IOPORT and INTVEC is set. If IOPORT and INTVEC are not set by the application, the synchronous driver may use them to store the I/O port and interrupt vector it chooses. Appendix C details the use of IOPORT and INTVEC with current DEVQUAL values.

Calling Preprocess

Having allocated and partially initialized the InterfaceRec, the application next calls Preprocess. As with all driver routines and service routines, the address of the InterfaceRec is passed as a parameter to Preprocess. Inside Preprocess, the driver examines the device qualifier and completely establishes device identity. It also looks at the synchronous parameter selections and makes sure that the hardware in use supports all of the selected options. Preprocess returns a result code to indicate whether or not it detects an error in the selected options, or to invoke the RECALL facility.

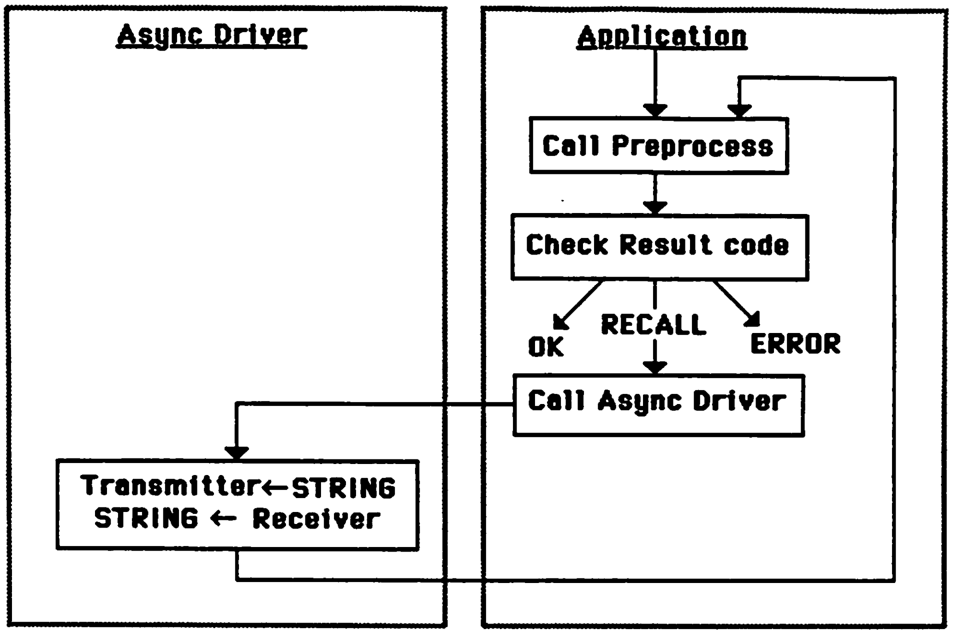


RECALL Facility

In general, Preprocess does not interact with the synchronous device to perform its functions, since the device may be operating in an alternate mode or capability which must not be disturbed. However, Preprocess may need to interact with the device in order to identify it or preset some of the synchronous protocol options. At such times, it is best to use an alternate driver (such as an asynchronous driver) to operate the device. To support this type of interaction, the RECALL facility is provided.

The RECALL facility lets Preprocess send a message to the device and receive a reply. To use the RECALL facility, Preprocess stores its message (a character string) in the InterfaceRec field STRING, then returns to the calling application with the result code RECALL. When it receives this RECALL result code, the application interacts with the alternate driver to communicate the message to the device and collect the response. The response, also a character string, is stored back into the STRING field, and Preprocess is called again.

With the Hayes Smartmodem 2400B, asynchronous AT commands must be issued to preset some of the synchronous protocol options. Preprocess stores the necessary command string in the STRING field and returns RECALL. The application then uses an asynchronous driver to issue the command string to the Smartmodem 2400B, collects the modem's response characters, and stores them back into the STRING. It then recalls Preprocess. This RECALL mechanism may be repeated before Preprocess returns a result code indicating that the process is completed, with or without error.



Preprocess Result Codes

Appendix B describes how Preprocess is called and how the result code is returned. The following result codes may be returned by Preprocess:

-	OK	Preprocess operations complete, no error.
-	RECALL	Application should transmit message in STRING to device, store reply back into STRING, and call Preprocess again.
-	BADQUAL	Driver does not understand the supplied device qualifier.
-	BADCALL	Preprocess called again;RECALL not activated.
-	MODENS	Selected SDLC/HDLC or BISYNC mode not supported by device.
-	WRAPNS	Wrap option (explained in Chapter 2) not supported by device.
-	METHDNS	Device doesn't support synchronous method implemented by the selected driver.

Calling StartSync

To actually start synchronous operation, the application calls StartSync. Before calling StartSync, the application prepares its side of the data exchange interface (see Chapter 2) since data transfer could commence immediately. Inside StartSync, the driver takes control of the device, programs its registers for the selected synchronous protocol options, installs an interrupt service routine, and begins operation. When control returns from StartSync, synchronous communication is fully operational.

CHAPTER 2--RECEIVING AND TRANSMITTING: SDLC/HDLC

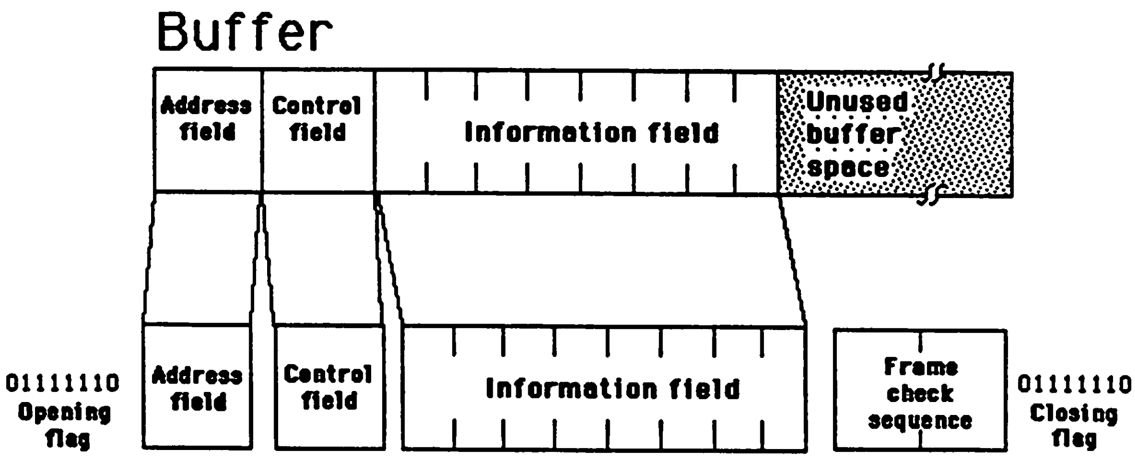
SDLC and HDLC

For our purposes, SDLC and HDLC are essentially the same. They share the same frame format, flag pattern, and frame check sequence. They both make use of techniques such as zero bit insertion and deletion. The differences between SDLC and HDLC lie at the level of data link control procedures. They differ in how messages relate to each other, not how individual messages are transmitted. Since the synchronous driver is concerned only with transmitting and receiving individual messages, and not with determining the correct response to a received message, it is not aware of the differences between SDLC and HDLC.

Frame Format

SDLC/HDLC messages are called frames. Each frame is stored in a separate buffer provided by the application. Frames cannot span buffer boundaries, and the maximum buffer size is 64K bytes. HSI assumes that 8-bit characters are being used, and provides no support for alternate character sizes.

The address and control fields of each frame are stored in the buffer as if they were data. In general, the synchronous driver does not distinguish between the address, control, and information fields of a frame. (An exception is the address detect option described later in this chapter.) Flags are not stored in the buffer, nor is the Frame Check Sequence (FCS).



SDLC Frame Format

What the Synchronous Driver Provides

The synchronous driver (with the hardware device) automatically appends the opening and closing flags to transmitted messages, and computes and appends the FCS. On the receive side, the opening and closing flags are detected, and the FCS is computed and checked. An error is reported if the received FCS is incorrect. Abort sequences are properly detected on receive data and properly generated on transmit data. Zero-bit insertion on transmit data and deletion on receive data is performed in a manner transparent to the application.

Synchronous Protocol Options for SDLC/HDLC

The application must select certain protocol options during set-up (before calling Preprocess). To makes its choices, the application sets the InterfaceRec fields (in the Synchronous Parameter Selection section) to the appropriate values. The choices relevant to SDLC/HDLC mode are as follows:

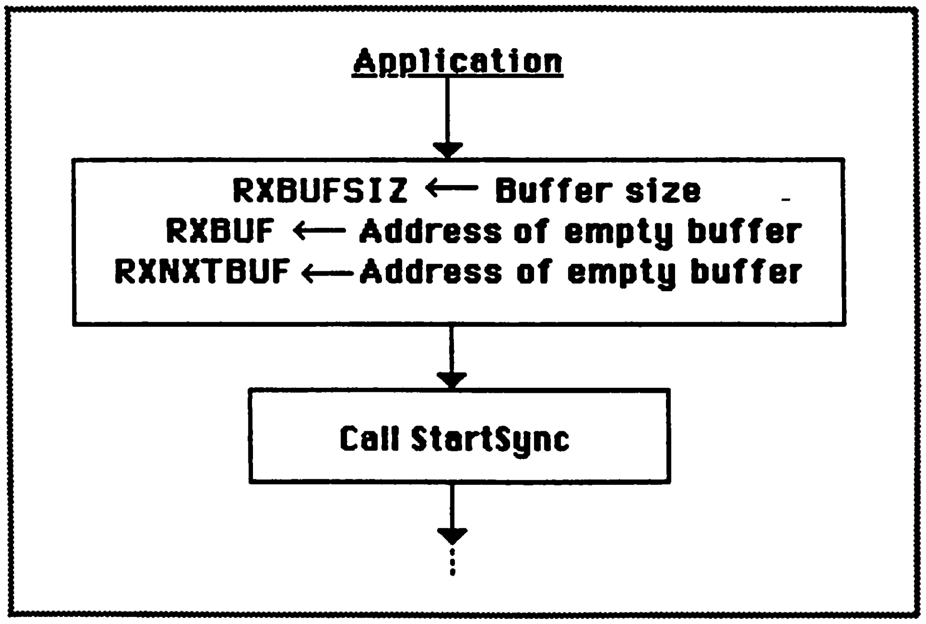
- BSC	To select SDLC/HDLC mode, this field is set to 0. Some synchronous devices might not support SDLC/HDLC operation; in such a case Preprocess would return the result code MODENS (Mode Not Supported).
- ADDRDET	To enable the address detect option, this field is set to 1; otherwise, it is set to 0. If the address detect option is enabled, only frames whose address field matches the programmed station address will be received and passed on to the application. Frames with the broadcast address FF will also be received. Frames that match neither address will be ignored and no error code will be generated. Extended address fields are not supported; only the first byte of the address field is checked.
- SDLCADDR	If the address detect option is enabled, the desired station address is stored in this field. If the option is not enabled, the driver ignores this field.
- BSCSYNC	The driver ignores this field for SDLC/HDLC mode.
- NRZI	If Non-Return To Zero Inverted (NRZI) encoding and decoding is desired, this field should be set to 1; otherwise, it is set to 0.
- FLAGIDLE	If this field is set to 1, the transmitter will send continuous flag patterns (01111110) when it has no data to transmit. If this field is set to 0, the transmitter will send a continuous mark level (all ones).
- WRAP	Some hardware devices have a wrap mode in which transmit data is immediately fed back into the receiver for test purposes.

To select wrap mode, this field is set to 1; otherwise, it is set to 0. If the hardware device in use does not support wrap mode, Preprocess will return the result code WRAPNS (Wrap Not Supported). If wrap mode is selected, it remains in effect for the duration of the synchronous session. To turn wrap mode off, a new synchronous session must be started by clearing the WRAP field, changing necessary synchronous parameters, and calling Preprocess and StartSync again.

Start-up Considerations for Receiver

The application and the synchronous driver go through the following sequence to receive incoming data. Before starting synchronous operation, the application stores the size of the buffers being used in the field RXBUFSIZ found in the data exchange section of the InterfaceRec. The buffer size selection is fixed at this point; it may be changed only after terminating synchronous operation. Notice, however, that RXBUFSIZ only pertains to the receive side.

Before starting synchronous operation, the application gets two empty buffers and places their addresses in RXBUF and RXNXTBUF. RXBUF points to the buffer which will store the next incoming message; RXNXTBUF points to the buffer which will store the next message.

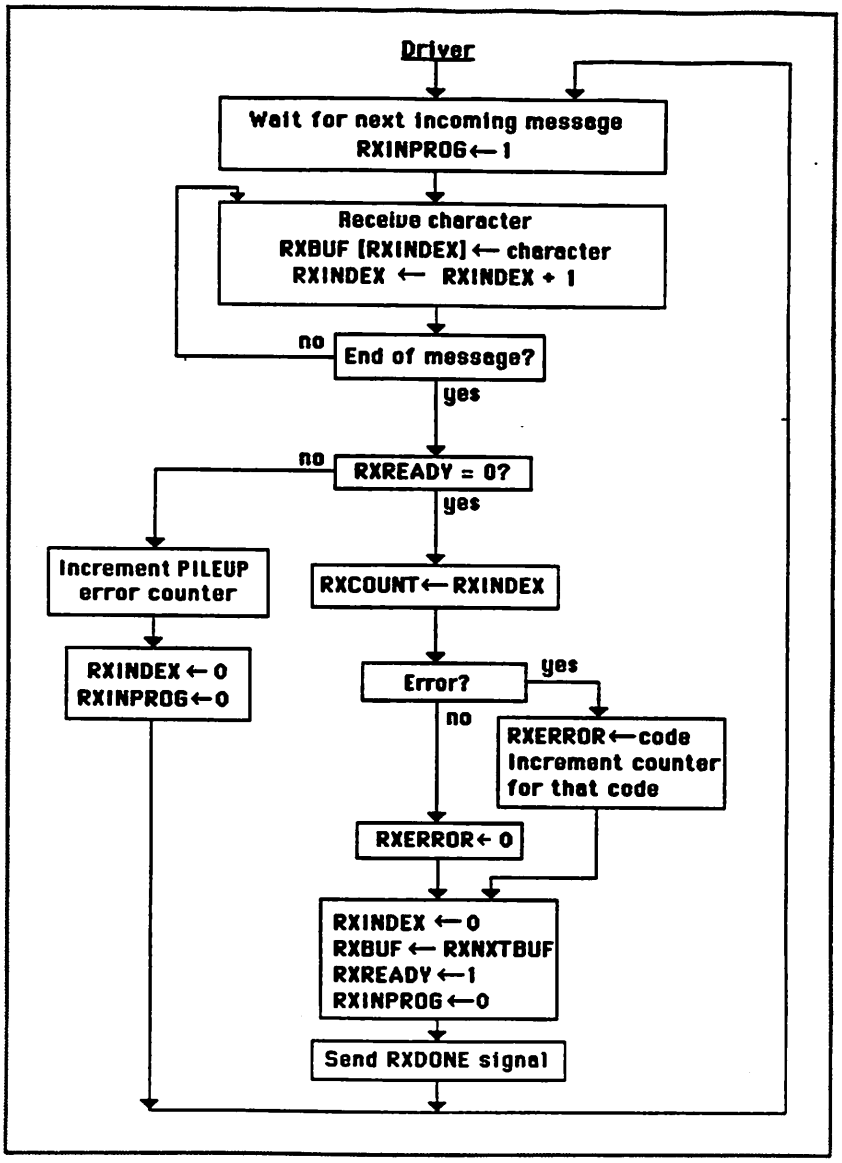

Application
RXBUFSIZ ← Buffer size
RXBUF ← Address of empty buffer
RXNXTBUF ← Address of empty buffer
Call StartSync

Receiver Operation: Driver

When the next message arrives, its characters are stored in sequential positions in the buffer pointed to by RXBUF. The driver uses the RXINDEX field to point to the next place to store a character in the buffer.

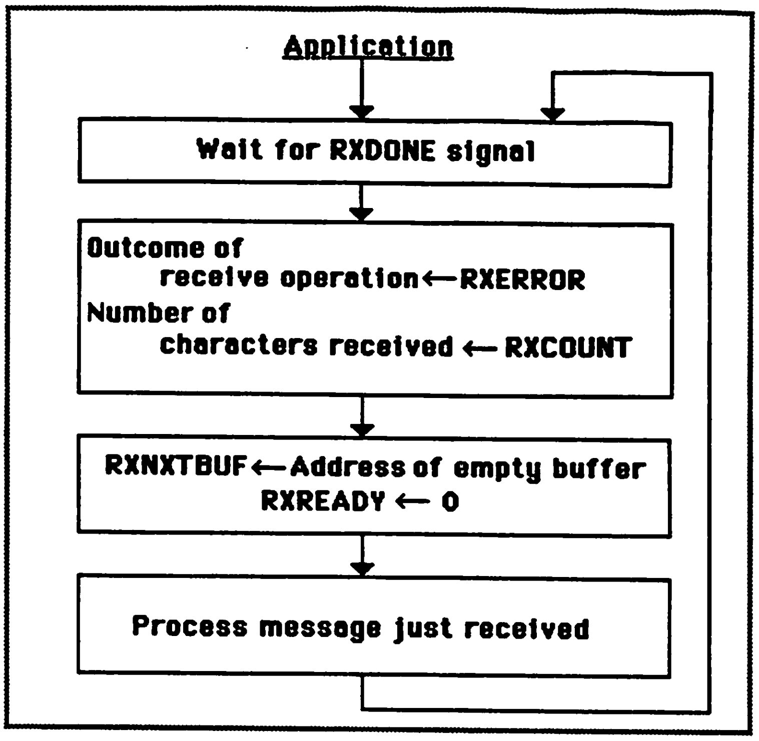
At the end of the message, the driver stores the number of characters received (the final value of RXINDEX) in the field RXCOUNT (the PILEUP error is explained later). If an error occurred, it stores an error code in RXERROR and increments the corresponding error counter. If no error occurred, the driver stores a zero in RXERROR. In preparation for receiving the next message, it then clears RXINDEX and transfers the buffer address in RXNXTBUF to RXBUF. Finally, the driver sets RXREADY to 1 and sends a signal, RXDONE, to the application.

The driver sets the flag RXINPROG to 1 when it receives the first character of a message, and clears it to 0 when the end of the message is reached. RXINPROG indicates that a receive operation is in progress.



Receiver Operation: Application

In response to the RXDONE signal, the application checks RXERROR and reads the number of characters received from RXCOUNT. Then it stores the address of a new, empty buffer in RXNXTBUF and clears RXREADY.


Application
Wait for RXDONE signal
Outcome of receive operation←RXERROR
Number of characters received ← RXCOUNT
RXNXTBUF←Address of empty buffer
RXREADY ← 0
Process message just received

PILEUP Errors

Notice that the receiver sets itself up to receive the next message immediately after the preceding message is received. Thus, the application has one entire message time to respond to the RXDONE signal. If for some reason the application fails to absorb the preceding message when the next message is received, the driver will find that the RXREADY flag is still set to 1. This situation is called a message PILEUP error.

To handle a PILEUP error, the driver increments the PILEUP error counter, clears RXINDEX to zero, and begins receiving the next message. The error is not reported in RXERROR because this field (and RXCOUNT) must not be changed once the RXDONE signal has been sent. The application will learn about the PILEUP error when it receives an out-of-sequence message (due to dropping one or more messages) or when it scans the error counters directly.

Active/Idle Status Reporting

Some applications require a response to a "line idle" condition. An idle condition occurs when eight or more consecutive 1 bits are detected on the line. Usually, the flag idle option is selected, in which the transmitter sends continuous flag patterns (01111110) between frames, preventing an idle condition. Detecting an idle condition may indicate that the transmitting station is out of order.

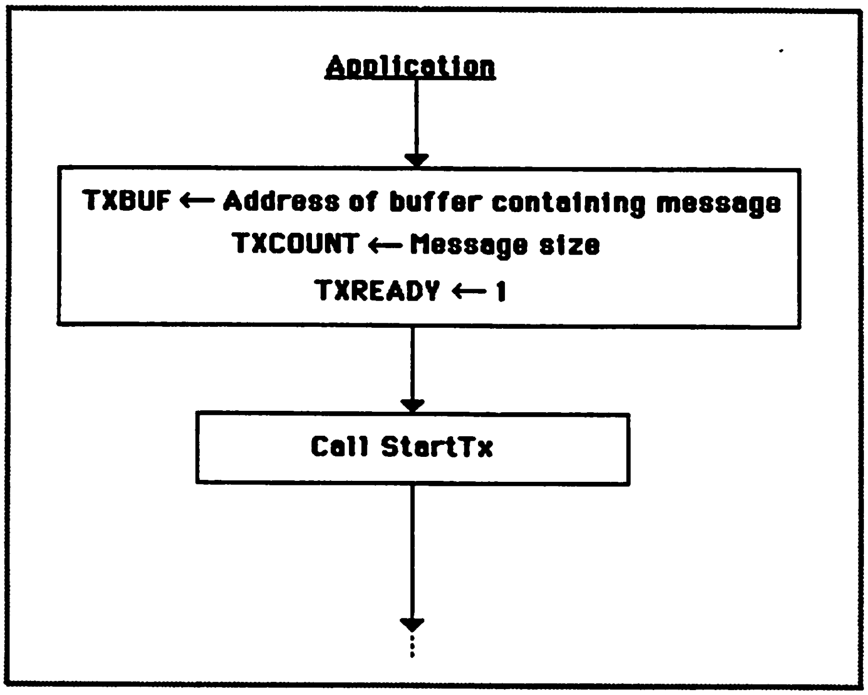
The driver (with the hardware in use) automatically maintains a flag in the InterfaceRec called RXACTIVE. This flag is set to 1 whenever the line is active and 0 when it is idle. Each time the RXACTIVE flag changes value, the driver sends the ACTCHG signal to the application. Using this mechanism, the application can maintain a time-out on the length of time the line is idle. If the idle condition is irrelevant to a given application, that application may ignore the RXACTIVE flag and the ACTCHG signals.

Start-up Considerations for Transmitter

On the transmit side, the application does nothing special before starting synchronous operation. Zeroing the InterfaceRec clears the transmitter's control fields, causing it to wait until the application has a message to transmit.

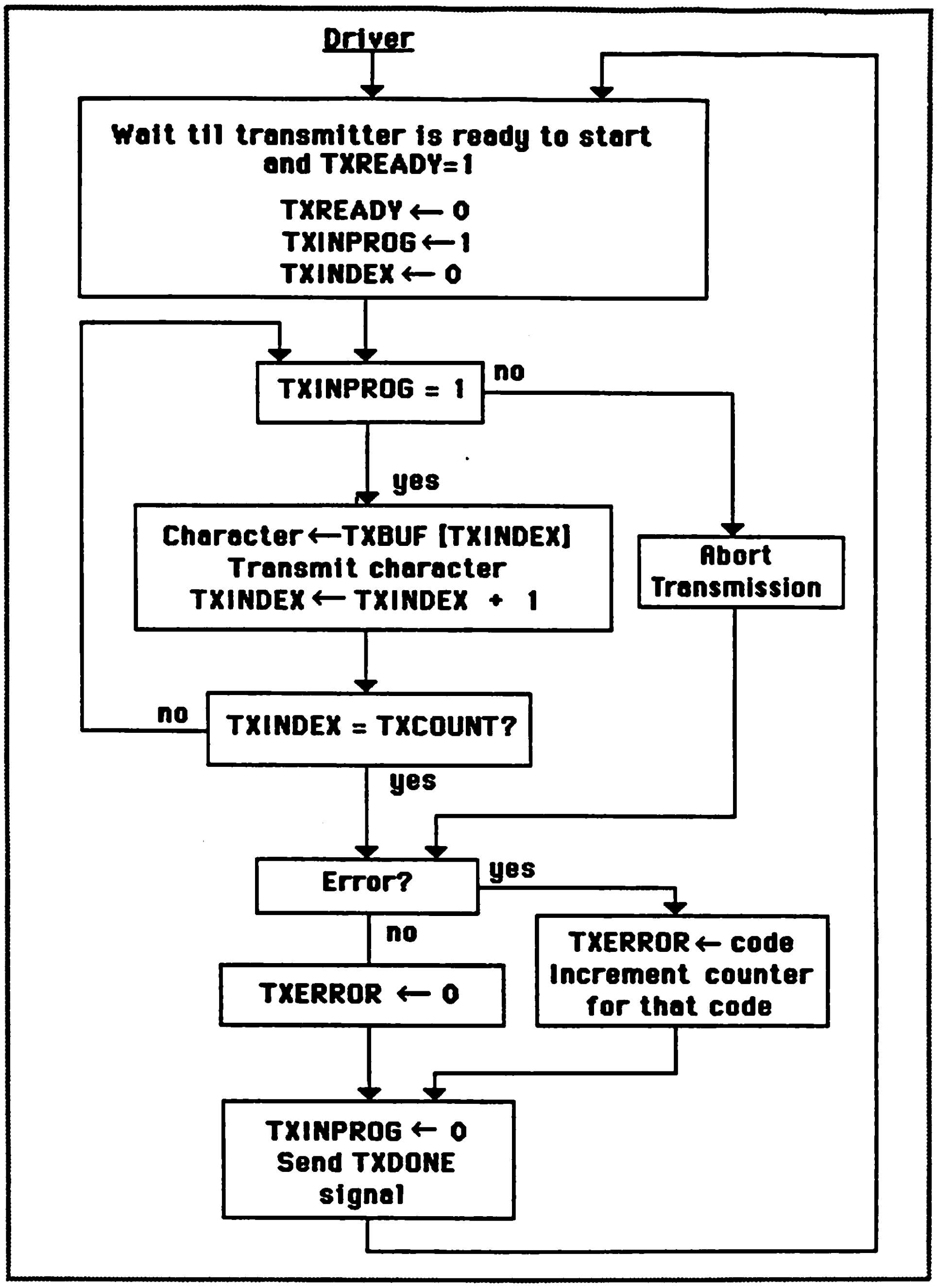
Transmitter Operation: Application

When the application has a message to transmit, it stores the address of the buffer containing the message in the field TXBUF. It stores the number of characters in the message (not the total size of the buffer) in TXCOUNT and sets TXREADY to 1. To make sure the driver notices that the application has given it a message to transmit, the application calls StartTx.

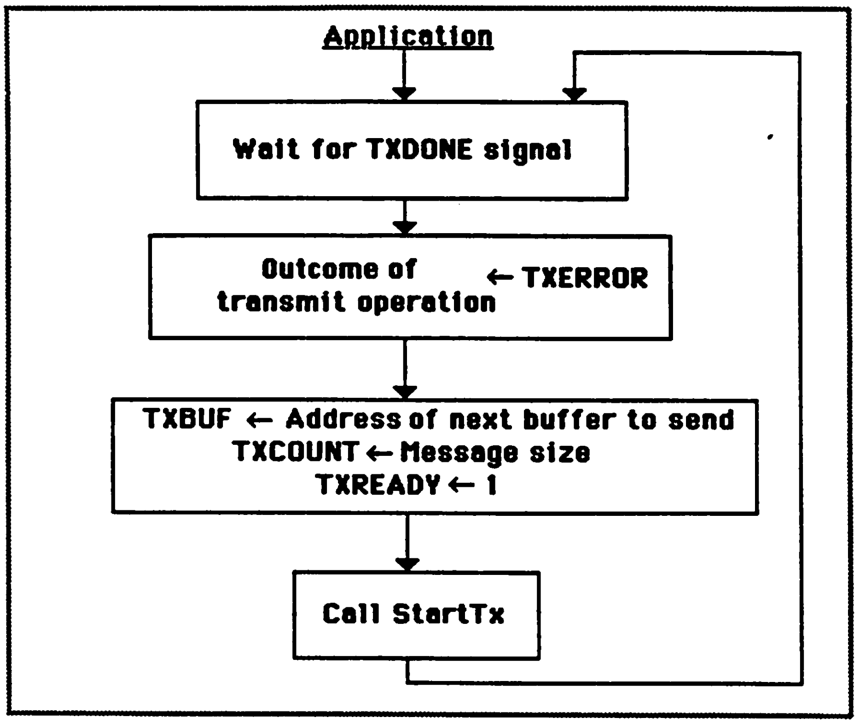


Transmitter Operation: Driver

When the hardware transmitter is ready to begin transmitting another message, the driver clears TXREADY, sets TXINPROG, and begins transmitting the message. It uses TXINDEX to keep its place in the buffer. When message transmission is completed, the driver stores zero (or an error code) in TXERROR, clears TXINPROG, and sends the signal TXDONE to the application. (The use of the TXINPROG flag is explained later.)



Upon receiving the TXDONE signal, the application checks TXERROR, then sets up the next message for transmission if there is one. The application should try to respond to the TXDONE signal as quickly as possible. This keeps the transmitter busy during periods of heavy data traffic.

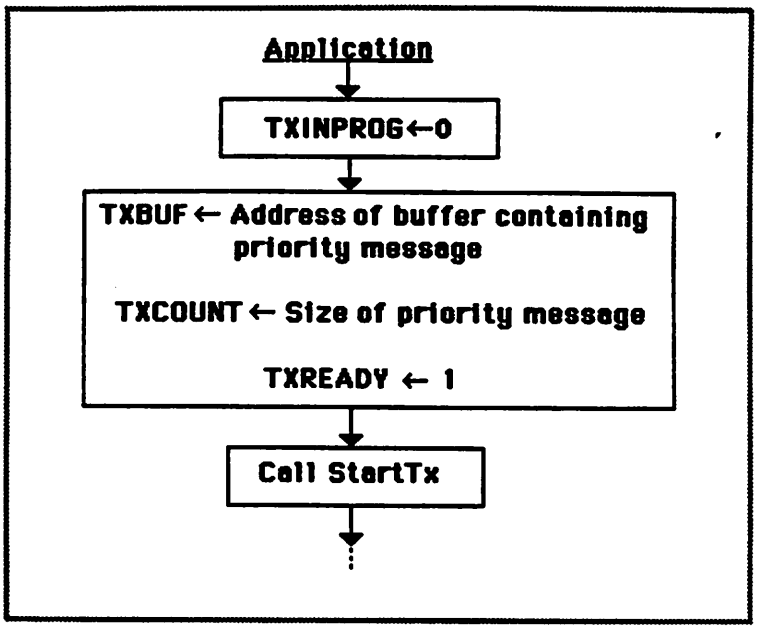


Aborting Transmissions

To prematurely abort a transmit operation (e.g., for a priority message), the application uses TXINPROG "transmission in progress" flag.

The driver sets TXINPROG at the beginning of each transmission and clears it at the end of each transmission. During the transmission, the driver checks TXINPROG before transmitting each character. If TXINPROG is still 1, transmission continues; if the application has cleared TXINPROG to 0, the driver aborts the transmission.

For example, suppose the application has a higher-priority message to transmit, requiring the current transmission to be aborted. The application would clear TXINPROG, set up the new message for transmission, then set TXREADY and call StartTx.


Application
TXINPROG←0
TXBUF ← Address of buffer containing priority message
TXCOUNT ← Size of priority message
TXREADY ← 1
Call StartTx

The following situation is possible: the application decides to abort the current transmission, but before it clears TXINPROG the current transmission is completed normally. In this case, the driver would have already cleared TXINPROG at the end of message transmission and no abort would occur.

Error Handling

The error code fields, RXERROR and TXERROR, report the outcome of each receive and transmit operation, respectively. When a transmit or receive operation is completed, the synchronous driver sets the error field to zero if no error occurred, or to a non-zero error code if an error did occur. The driver also increments the error counter of the corresponding error code every time an error occurs, to provide a more permanent record of errors. There are separate error counters for the transmit side and the receive side.

The RXERROR and TXERROR fields can only be used when an error occurs during a specific transmit or receive operation. The error counters record errors which occur both during and outside of transmit and receive operation.

Error Codes

HSI provides for 20 error codes, numbered 1 through 20. (Code 0 is used to indicate that no error occurred.) Six error codes, numbered 1 through 6, are defined by HSI. The remaining 14 are reserved for future or specialized use by each application. Chapter 3, on BISYNC operation, describes how the undefined error codes may be used. There is an error counter in the InterfaceRec for the 20 error codes for both transmit and receive, (40 altogether).

The error conditions and their code numbers are:

1.	OVUNDR	Hardware overrun/underrun.
2.	ABORT	Abort sequence transmitted or received.
3.	NOSIG	Modem signal lost (DCD for receive side, CTS for transmit).
4.	PILEUP	Reception of next message completed before application processed previous message.
5.	OVFLOW	Incoming message overflowed buffer (RXLIMIT was less than message length).
6.	BADCK	Incoming message has bad check sequence.

Internal Errors

Two fields, RXINTERR and TXINTERR, may be used by the synchronous driver to report internal errors. An internal error usually occurs when the driver is in the wrong operating mode (e.g.,the driver may be in the end-of-message mode before the first character of a message has been received).

HSI does not specify the format of the error codes stored in RXINTERR and TXINTERR, beyond stipulating that they be nonzero. A zero code indicates no error. Internal error should not occur; it indicates a programming error in the synchronous driver or a problem in the hardware.

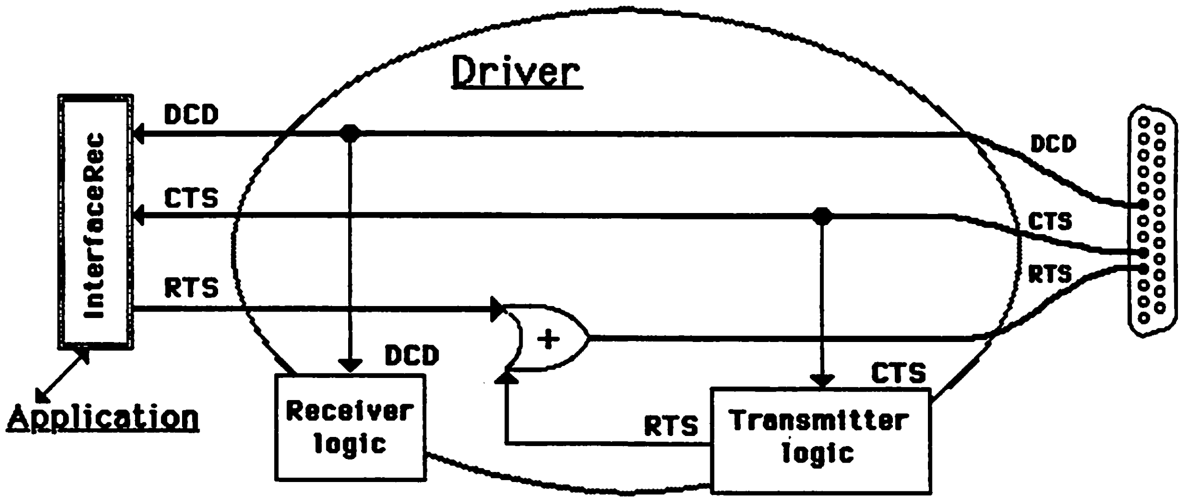
Modem Signal Usage

HSI does not define specific modem signal usage, except for the signals RTS (Request To Send), CTS (Clear To Send), and DCD (Data Carrier Detect). Assuming the hardware being used supports them, these three signals are controlled or tested by the synchronous driver during the course of transmit and receive operations. The driver requires the presence of an active DCD signal before it validates received data. If DCD is not present or is lost during message reception, an error code is generated.

On the transmit side, the synchronous driver raises RTS each time it is ready to transmit a message, then it waits for CTS to go high before beginning message transmission. At the end of the message, it drops RTS. If CTS is lost during message transmission, an error code is generated.

The application can program RTS high or low, just like any other modem signal that functions as an output. The state of RTS as outputted is the logical OR of the application's setting and the driver's control. If the application programs RTS low, the signal will be under the control of the driver, and will go high and low as messages are transmitted. If the application programs RTS high, it will be fixed high. In every case, however, the driver requires an active CTS signal for proper message transmission.

In certain types of hardware, other modem signals (such as Data Set Ready, DSR) might be required in order for transmit and receive operations to occur. HSI does not address these signals.



CHAPTER 3--RECEIVING AND TRANSMITTING: BISYNC

BISYNC vs. SDLC/HDLC

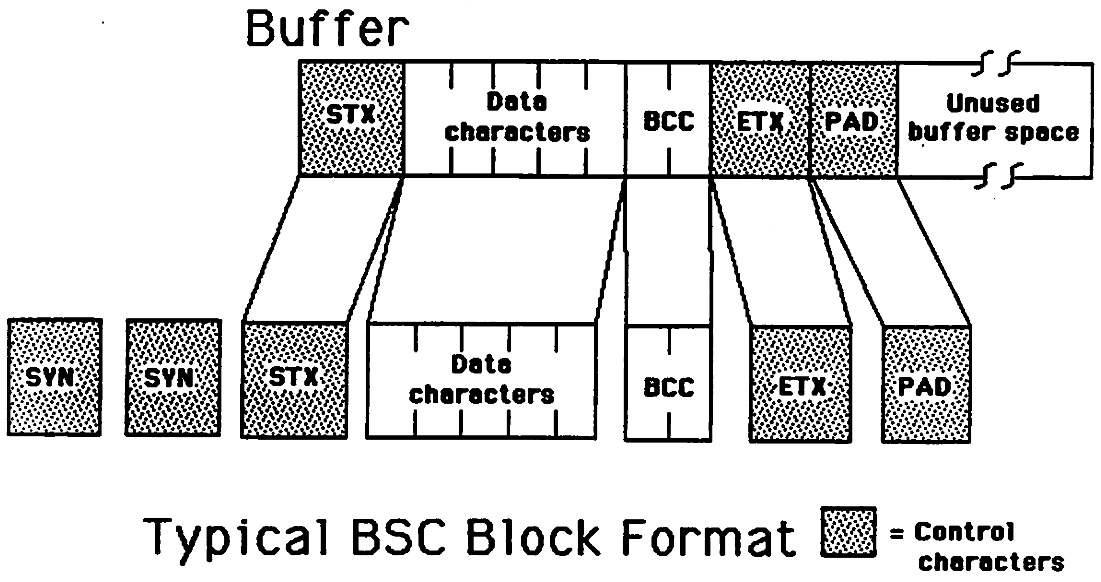
BISYNC operation is identical to SDLC/HDLC operation in many ways. The same mechanism is used to exchange buffers containing messages back and forth between the synchronous driver and the application. All of the fields in the data exchange interface section of the InterfaceRec work the same way for BISYNC mode as they do for SDLC/HDLC mode. Refer to Chapter 2, covering SDLC/HDLC operation, for a complete discussion of the topics that are common to both BISYNC and SDLC/HDLC operation. This chapter points out the differences between the two modes of operation and covers topics that pertain only to BISYNC operation.

Block Format

BISYNC messages are called blocks. Unlike SDLC/HDLC frames, which have a very well-defined and universally accepted format, BISYNC blocks come in many different forms. There is no universally accepted format for BISYNC blocks. For example, it is not possible to define a method for establishing the end of a BISYNC block that will work for every variation of the BISYNC protocol in use today. Thus, BISYNC operation is characterized by a greater reliance on the application to participate in the operation of the synchronous data link.

A BISYNC block consists of a series of characters. Blocks always include one or more control characters and may also include data characters. Each block is preceded by a series of two or more synchronization (SYN) characters. The purpose of these characters is to lock the receiver into character synchronization. Blocks containing data characters usually also contain a Block Check Character (BCC) sequence, analogous to the SDLC/HDLC's Frame Check Sequence. Each block is stored in a separate buffer and cannot span buffer boundaries. The maximum buffer size is 64K bytes. Eight bit characters are assumed, but any character set can be used.

All characters comprising the block, except for the opening SYN characters, are stored in the buffer. Any SYN characters appearing later in the block, as well as all other control, data, and block check characters, are stored in the buffer.



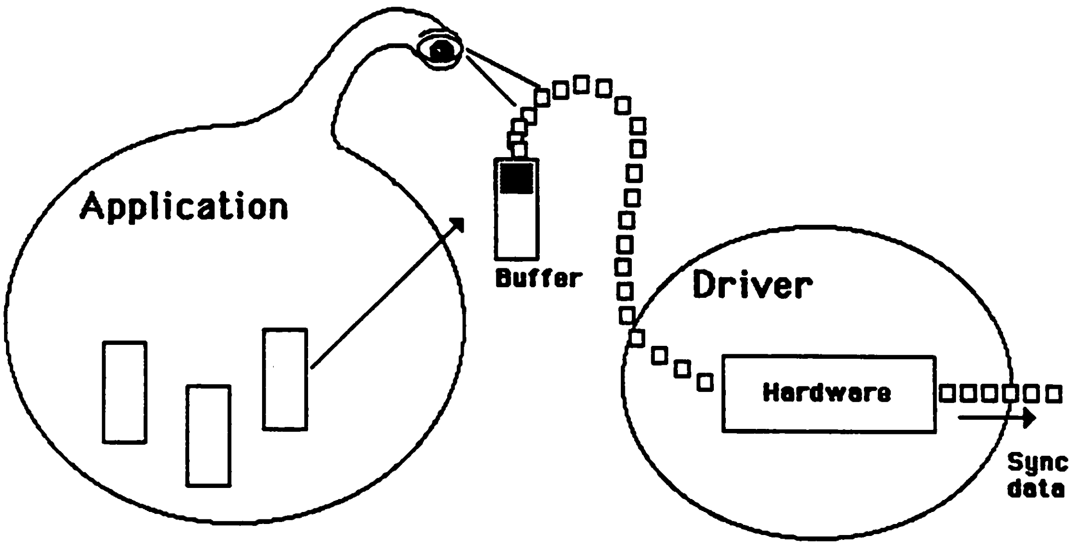
What the Synchronous Driver Provides

The synchronous driver (with the hardware being used) automatically appends two opening SYN characters to transmitted blocks. The application may specify the particular 8-bit character which will serve as the SYN character. On the receive side, the opening SYN characters are detected but not stored in the buffer. The application processes block check characters, control characters, and idle SYN characters. The application also determines the end of the message being received.

Per-character Processing

In order to perform its functions, the application must get involved in the character-by-character processing of data. The two per-character service routines, TxCharProc and RxCharProc, are defined to enable this. The presence of per-character processing does not eliminate the data-buffering function of the driver.

The synchronous driver continues to transfer data between the buffers and the hardware, but in BISYNC modes, the driver lets the application "peek" at the data as it is transferred and give the driver special character-by-character processing instructions.



Synchronous Protocol Options for BISYNC

The following are the synchronous protocol choices relevant to BISYNC mode:

-	BSC	To select BISYNC mode, this field is set to 1. Some synchronous devices might not support BISYNC operation; in such cases, Preprocess would return the result code MODENS (Mode Not Supported).
-	ADDRDET	The driver ignores this field for BISYNC mode.
-	SDLCADDR	The driver ignores this field for BISYNC mode.
-	BSCSYNC	The application stores the desired SYN character in this field. (usually 32 hex for the EBCDIC character set or 16 hex for ASCII).

-	NRZI	If Non-Return To Zero Inverted (NRZI) encoding and decoding is desired, this field should be set to 1; otherwise, this field is set to 0.
-	FLAGIDLE	If this field is set to 1, the transmitter sends continuous SYN characters when it has no data to transmit. If this field is set to 0, the transmitter sends a continuous mark level (all ones). Notice that this choice is made only once during set-up for the synchronous session (before calling Preprocess). It is not possible to switch back and forth between mark idle and SYN idle in the middle of one session.
-	WRAP	Setting this field to 1 selects the hardware wrap option, if available. See Chapter 2 for a more complete discussion of the wrap option.

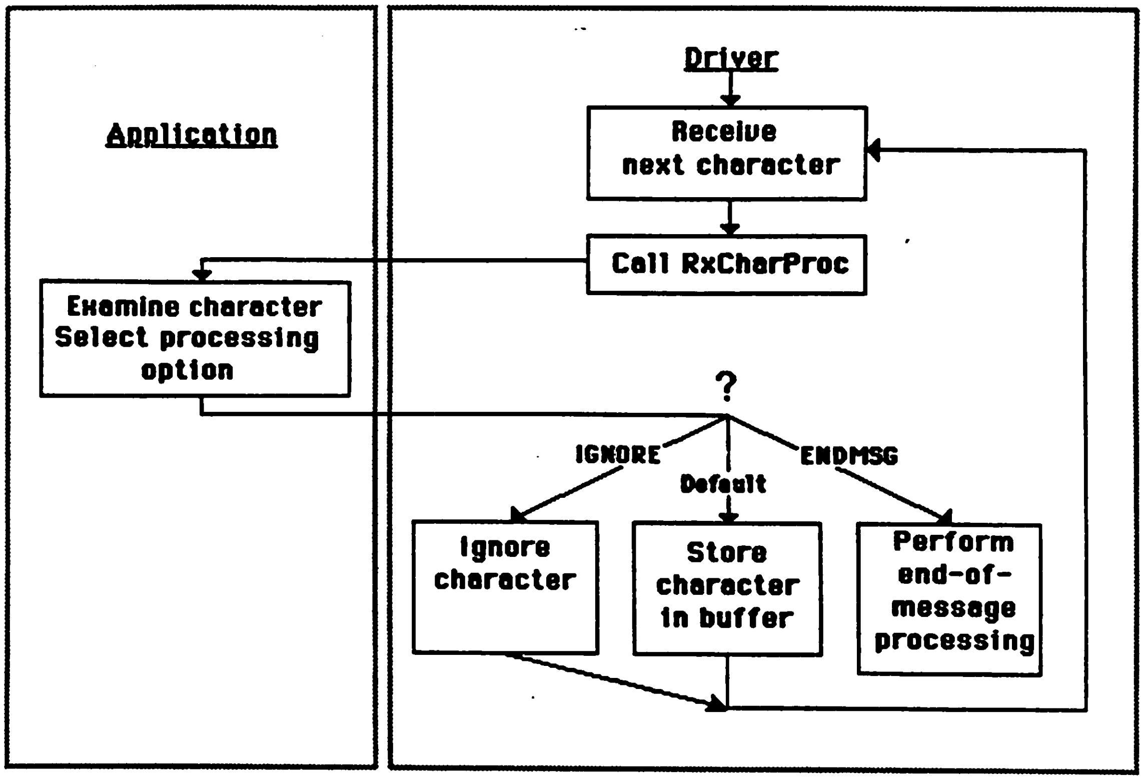
Receiver Operation

The application and the synchronous driver go through the same sequence of events for BISYNC operation as described in Chapter 2 for SDLC/ HDLC operation. At the same time, however, per-character processing occurs. The driver calls RxCharProc just after it receives each character. Inside RxCharProc, the application has the opportunity to examine the stream of incoming characters and influence the driver's processing by selecting certain processing options.

Unless instructed otherwise, the synchronous driver simply stores the character in the buffer and waits for the next character after control returns from RxCharProc. As an alternative, the application can select two select processing options on each received character:

-	IGNORE	Ignores the character just received.
-	ENDMSG	Treats character as the last character of the message and performs end-of-message processing. With this option, RxCharProc may return an error code to be associated with the message, otherwise the driver indicates no error.

Either or both processing options may be selected independently for each character.


Application
Driver
Receive next character
Call RxCharProc
Examine character Select processing option
?
IGNORE
Default
ENDMSG
Ignore character
Store character in buffer
Perform end-of-message processing

The use of per-character processing to implement BISYNC communications is the application programmer's decision. HSI requires only that the ENDMSG option be selected at the proper time to terminate each received message. To do this, the application implements some sort of finite-state machine to follow the BISYNC protocol for each message that it receives. Each time RxCharProc is called, the application uses the character just received to advance the machine to a new state. When a terminal state is reached, the ENDMSG option is selected.

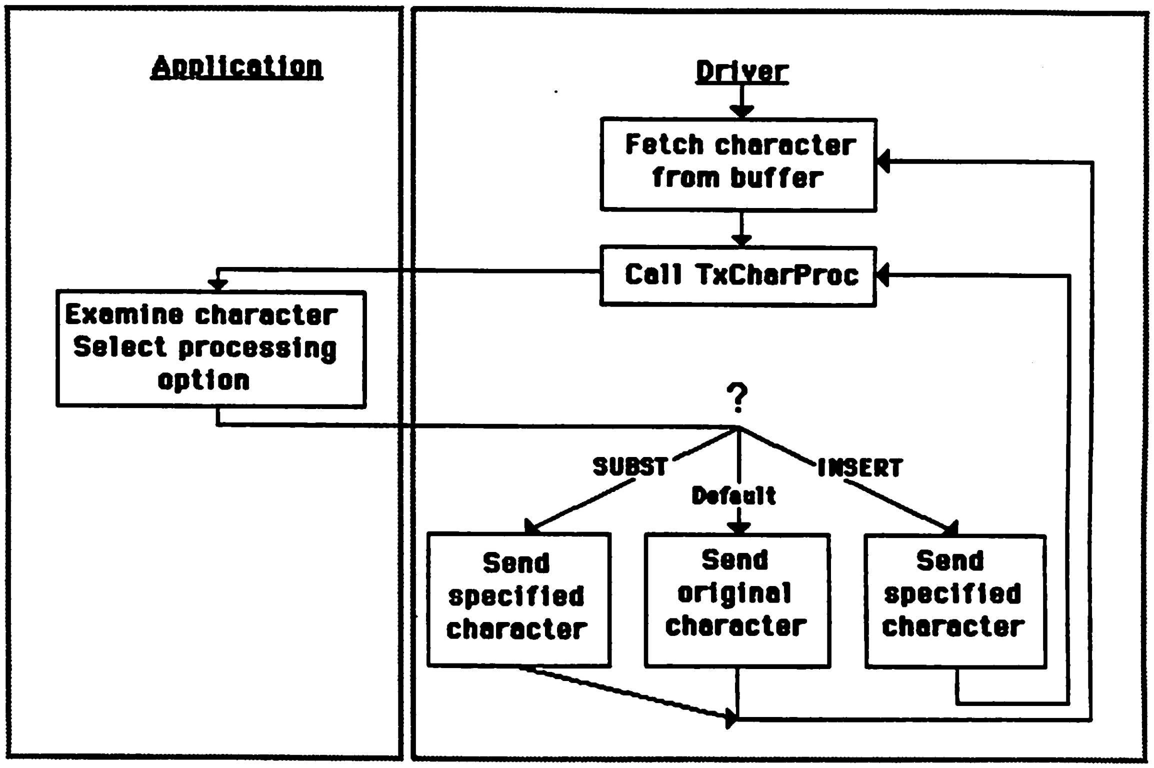
With per-character processing, the application can also compute and check the block check character sequence at the end of each message, and return an error code with the ENDMSG option if the check sequence is bad. Other error conditions, such as BISYNC protocol violations, can be detected and reported making use of the undefined error codes. With the IGNORE option, the application can strip idle SYN characters and DLE (Data Link Escape) characters from transparent text. By stripping these characters from the data before it is stored in the buffer, the data need not be recopied to remove the characters later.

Refer to Appendix B for a detailed description of how RxCharProc and TxCharProc are called, what parameters are passed to them, and what results they return.

Transmitter Operation

Transmitter operation in BISYNC mode is similar to SDLC/HDLC mode, with the addition of per-character processing. After the synchronous driver reads each character from the buffer, and before it transmits the character, the driver calls TxCharProc. Inside TxCharProc, the application can examine the stream of transmit data and select processing options. If no processing option is selected, the driver transmits the character normally. One of the two alternatives to this default processing may be selected:

1.	INSERT	By selecting this option, the application can cause a specified character to be inserted in the message just before the character about to be transmitted. After transmitting the character, the driver calls TxCharProc again for the character that was to be transmitted previously.
2.	SUBST	By selecting this option, the synchronous driver transmits a substitute character to replace the character about to be transmitted. After transmitting the substitute character, the driver proceeds to the next character in the buffer.



Per-character processing on the transmit side is not as critical to BISYNC link operation as it is on the receive side. HSI does not specify what or when processing options must be selected. A BISYNC link may be run without using per-character processing on the transmit side, by storing the message in the buffer exactly as it is to be transmitted.

Some applications may choose to use per-character processing on the transmit side to process data as it goes out to the transmitter, rather than processing data in advance in the buffer. For example, the trailing block check character sequence can be computed inside TxCharProc and placed in the message on top of dummy, place-holder characters using the SUBST option. Also, idle SYN characters can be inserted at regular intervals into the message by tying TxCharProc into some kind of real-time clock service provided by the application. DLE characters can be inserted into transparent text where necessary via the INSERT option. Again, a finite-state machine is usually used to control TxCharProc, so that it chooses each processing option at the right time.

Buffer Switching

The open-ended nature of HSI, with regard to BISYNC operation, allows for creative application designs. For example, it may be advantageous in some applications to store control characters separately from data characters, the latter being stored in data buffers. TxCharProc or RxCharProc could be programmed to switch the BUF, COUNT, and INDEX fields to point to different data sources as message transmission (or reception) progresses.

For example, TXBUF can be set to point to a data area containing a fixed sequence of control characters. TXCOUNT can be set to the length of this sequence, or to some arbitrary high value. Under the control of a finite-state machine inside TxCharProc , the synchronous driver is allowed to transmit the control characters. TXBUF, TXCOUNT, and TXINDEX then can be reset to point to a buffer containing the data portion of the message. To support this type of creative programming, the driver does not hold the value of variables such as TXBUF, TXCOUNT, or TXINDEX in registers across calls to TxCharProc or RxCharProc.

Modem Signal Usage

Modem signals are used exactly as described in Chapter 2 for SDLC/HDLC mode.

Error Handling

BISYNC error handling is the same as that described for SDLC/HDLC, except that user-defined error codes may be returned from RxCharProc via the ENDMSG option. If RxCharProc fails to return the ENDMSG option for a given incoming message, a buffer overflow error (OVFLOW) will eventually be generated.

Aborting Transmission and Reception

HSI does not support an abort sequence for BISYNC mode. If an abort sequence is defined for the BISYNC protocol variant being used in a given application, it is detected by RxCharProc. The ENDMSG option is taken, and the ABORT error code is returned. On the transmit side, the synchronous driver responds to the application's clearing the TXINPROG flag the same as it does in SDLC/HDLC mode, except that no abort sequence is sent. The transmitter simply reverts to its idle state (SYN idle or mark idle). Transmission of the next message may begin immediately thereafter.

Active/Idle Status Reporting

This facility is not relevant to BISYNC mode.

Finite State Machine Initialization

It may be simpler in some applications for the finite state machines within RxCharProc and TxCharProc to automatically reset themselves to their initial states at the beginning of each message rather than having this action performed at another time (when the state machines detect end-of-message, for example). To do this, RxCharProc and TxCharProc should test the InterfaceRec fields RXINDEX and TXINDEX, respectively, each time they are called. If the field has the value 1, the state machine should reset itself then process the first character of the message. Otherwise, the character should be processed normally.

CHAPTER 4--TERMINATING SYNCHRONOUS OPERATION

To terminate synchronous operation, the application calls EndSync. Inside EndSync, the synchronous driver shuts the device off, removes the interrupt service routine, and performs whatever other operations are necessary to terminate synchronous operation.

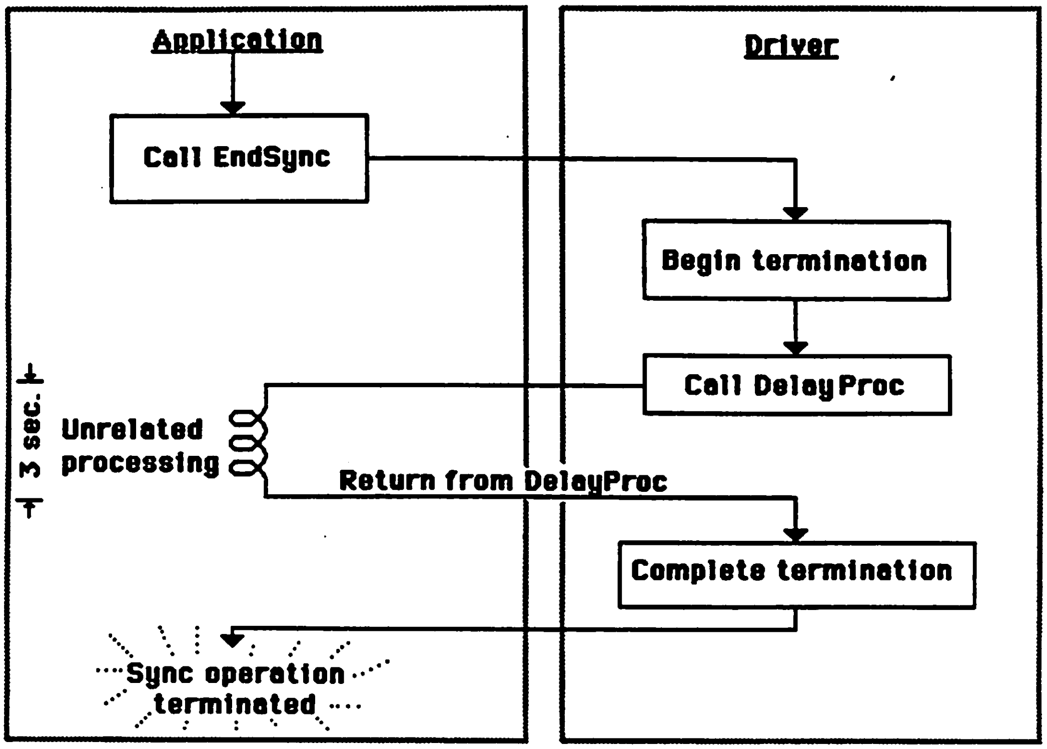
Effect on Operations in Progress

If transmit or receive operations are occurring when EndSync is called, the driver will abort them automatically. For transmit operations, an abort sequence is sent; for receive operations, the driver pretends to receive one. The appropriate signal, TXDONE or RXDONE, is sent with an ABORT error code. The signaling mechanism may be invoked during EndSync processing; it should be left intact on the application side until control returns from EndSync.

Use of DelayProc

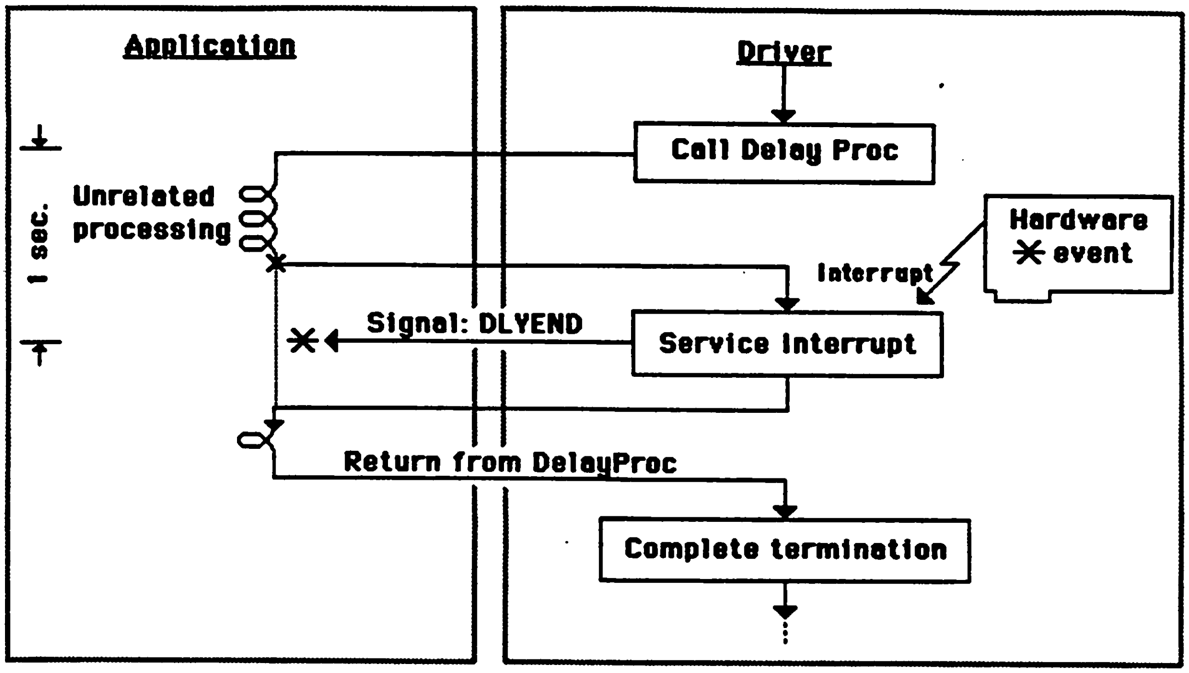
Some synchronous devices require a time delay during their deactivation sequence. This time delay is obtained by the driver calling the DelayProc service routine. The application may do whatever it wishes with the CPU during the delay interval, as long as it returns control when the interval expires. The use of DelayProc avoids the use of an idle loop inside the driver. DelayProc is called only during termination processing, as the result of a call to EndSync. It may be used by some device drivers and not others.

Since the delay required by synchronous devices may differ, the synchronous driver passes the desired time delay as a parameter to DelayProc. The delay may vary from 0 to 65535 milliseconds and should be timed as accurately as possible by the application using the timer services provided by the host operating system or programming language. However, the driver must not depend on this mechanism for extremely accurate delays.



Usually, DelayProc is used to time-out an event which occurs during termination. If DelayProc returns before the event occurs, the synchronous driver assumes that something is wrong and stops waiting for the event. If the event occurs before the time-out expires, the delay is no longer needed and DelayProc may return. A signal, DLYEND, is defined for use in this situation.

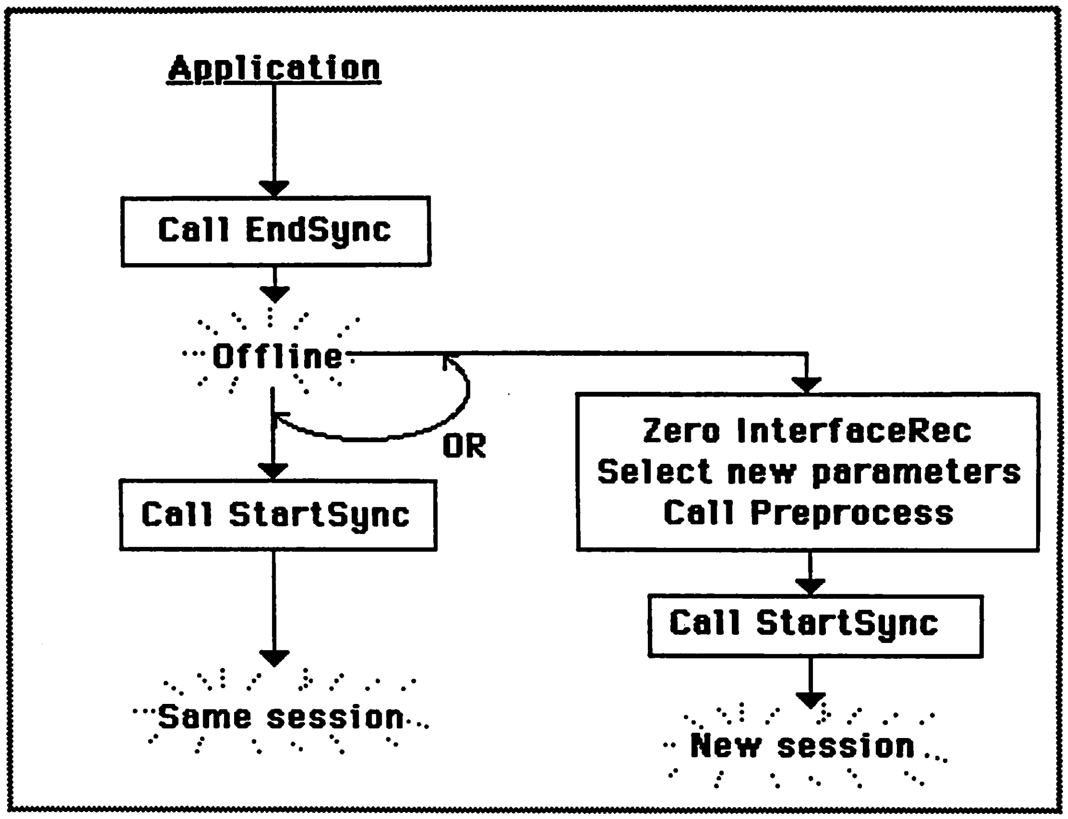
If DelayProc has been called to time-out an event within 3 seconds, and the event occurs after 1 second, the device hardware generates an interrupt which the synchronous driver services. The driver recognizes the event and sends a DLYEND signal to the application. At this point, interrupt servicing is completed and the application again has control of the CPU. In response to the DLYEND signal, the application causes DelayProc to return control immediately to the driver.



If a delay is not in progress when the DLYEND signal is received, the application may apply the signal to the next call of DelayProc, returning the call immediately. In fact, the DLYEND signal may set a flag which returns all subsequent calls to DelayProc immediately. This flag should be reset before StartSync is called again.

Restarting Synchronous Operation

By calling StartSync again, synchronous operation may be restarted after it has been terminated. The application should not call Preprocess again unless a new synchronous session (with a new set of parameters) is desired. Synchronous communications picks up where it left off when EndSync was called. Since most synchronous protocols have error detection and automatic request (ARQ) for re-transmission facilities, the service interrupt is not visible to the user except for a momentary pause in communications. An extended service interrupt, however, may result in a time-out by the other party and a termination of the call on that side.

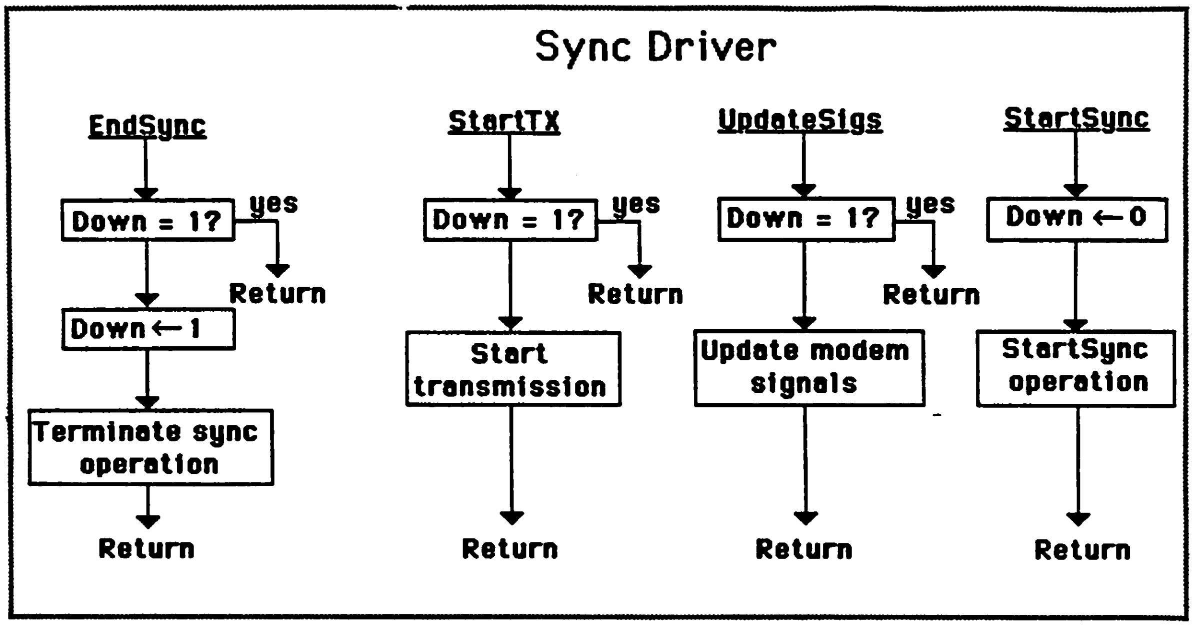


Protection Against Race Conditions

The synchronous driver routines EndSync, StartTx, and UpdateSigs have built-in protection against race conditions which might occur during termination. Since calls to these routines may be made from several different tasks running inside the application, improper calling sequences may occur during termination.

For example, one task might call StartTx after another task called EndSync. Two separate tasks might decide to terminate synchronous operation, resulting in two consecutive calls to EndSync. Even though these situations are programming errors, they may be difficult to avoid in complex applications.

A simple mechanism built into EndSync, StartTx, and UpdateSigs prevents these errors from occurring. The first call to EndSync sets an internal flag indicating that the driver is down. When this flag is set, a subsequent call to any of these three routines will result in no operation. The flag is reset the next time StartSync is called.

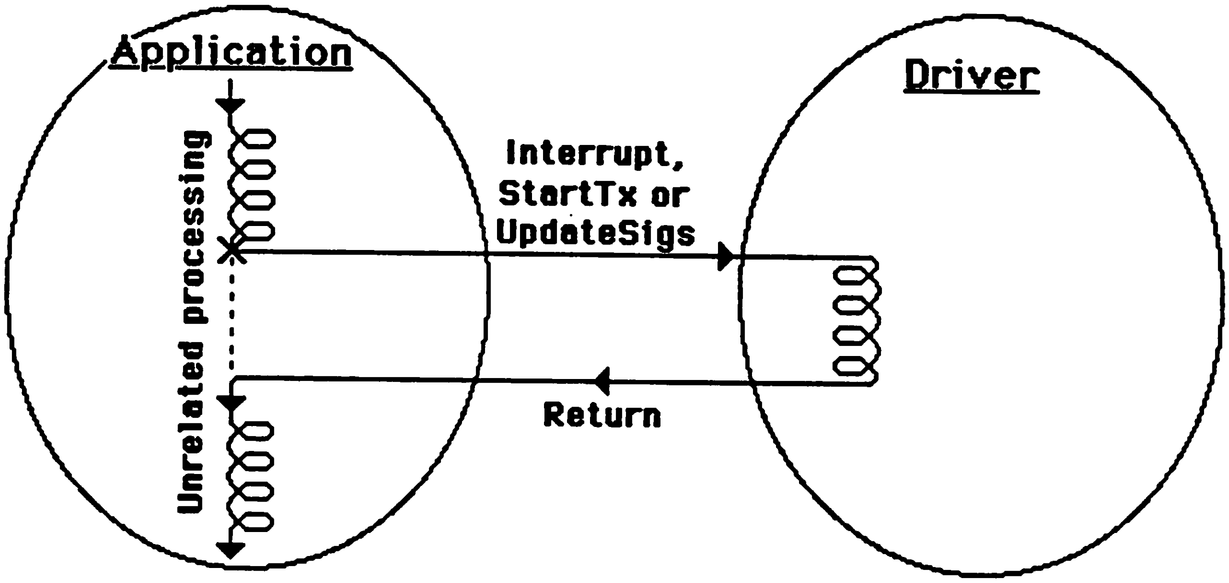


CHAPTER 5--SIGNALING MECHANISM

Application and Synchronous Driver Processes

The operation of the application and the synchronous driver may be considered two independent processes. Once synchronous operation begins, the synchronous driver runs independently from the application, getting control of the CPU in response to device interrupts. Exceptions occur when the application explicitly calls a driver routine, (either StartTx or UpdateSigs). The driver can be thought of as always taking control from the application at unpredictable times, leaving the application in a suspended state while the driver executes.

When the driver completes all processing, it returns the CPU to the interrupted flow of control in the application. In a driver routine call, control is returned to the caller, but the application usually is unaware that anything happened.



Need for Signaling Mechanism

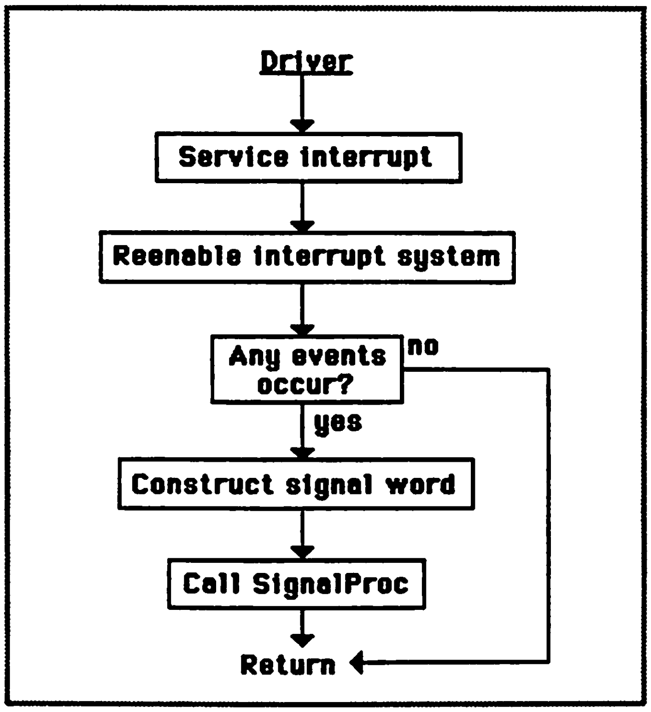
Some occasions during the execution of the synchronous driver require a reaction from the application. For example, if an incoming message has just been received, the application must attempt to process that message. If the driver simply returns control to the interrupted process, the application will continue as though no message was received.

Clearly, a mechanism must be defined to permit the driver to occasionally communicate with the application. This mechanism would be invoked when events occur within the driver, calling for the application to alter its flow of control. This is the role played by the signaling mechanism.

How the Synchronous Driver Sends Signals

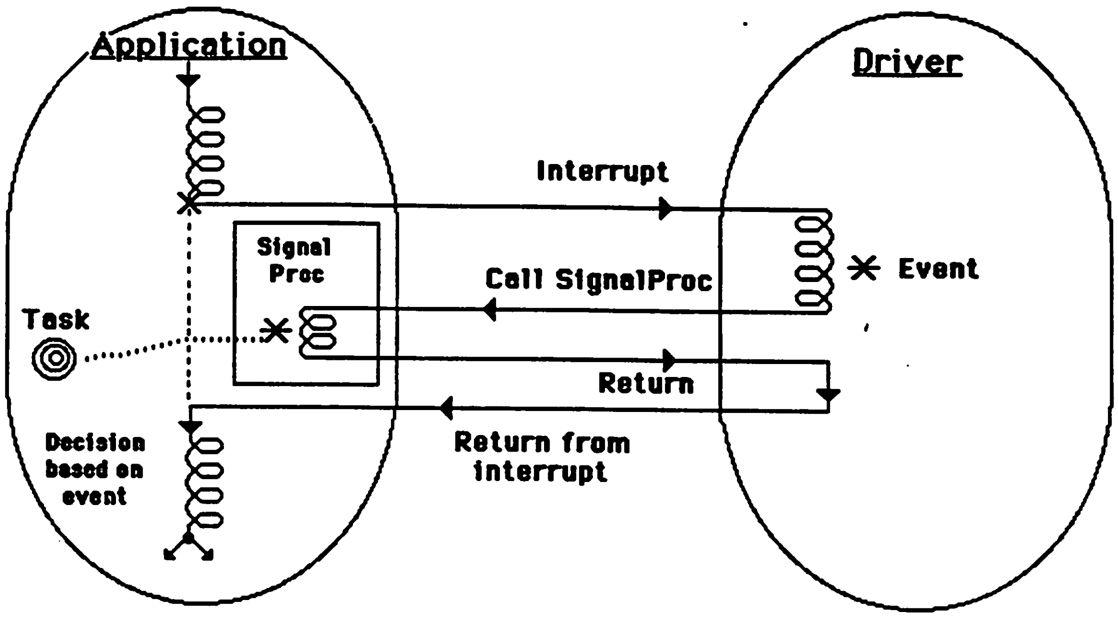
The signaling operation occurs just before the synchronous driver returns control to the interrupted process (or caller of StartTx or UpdateSigs). At this point, the driver's immediate processing needs have been satisfied, all interrupt conditions have been cleared, and the interrupt system has been re-enabled.

Before returning control to the interrupted process, an extra step is inserted: the driver constructs a special signal word, indicating the signals it wants to issue, and calls SignalProc, one of the five service routines provided by the application. The signal word is passed as a parameter to SignalProc.



How the Application Responds

Inside SignalProc, the application examines the signal word and reacts, based on the indicated signals. Sometimes all of the appropriate processing can be performed inside SignalProc. If a multitasking system is in use, SignalProc may activate one or more high-priority tasks to perform the processing. Eventually control returns to the synchronous driver from SignalProc and is then transferred back to the interrupted process.

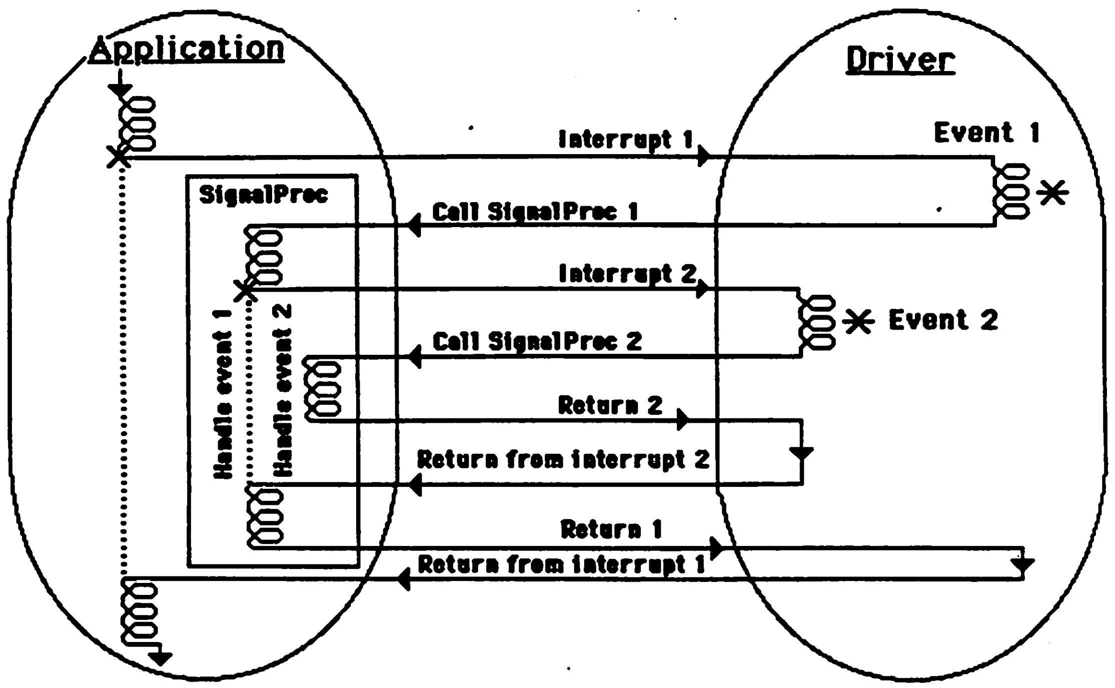


Nested SignalProc Calls

If a task activated by SignalProc has a higher priority than the one that was originally interrupted (which the multitasking system assumes is still in control), it may preempt the execution of SignalProc itself. In this situation the execution of SignalProc is suspended as long as it takes to execute the high-priority task. This task may in turn be preempted by even higher-priority tasks. A relatively long period of time may, therefore, elapse between a call to SignalProc and the corresponding return.

While SignalProc, or the tasks it activates, is executing, interrupts may occur resulting in new calls to SignalProc, even before control returns from the original call. These nested SignalProc calls may result in signals being processed out of order. This should cause no problem, since the signals do not interact with each other.

Because nested calls may occur, SignalProc must be reentrant, and process more important tasks (such as RXDONE and TXDONE) before less important ones.



The Signal Word

Five possible signals are mapped to individual bits in the 16-bit signal word. For each signal the synchronous driver wishes to send, the corresponding bit is set to 1; all other bits are set to 0. After constructing the signal word, the driver passes it as a parameter to SignalProc. Thus, any combination of the five signals can be sent in one operation. Appendix B describes in detail how SignalProc is called, including the format of the signal word.

The five possible signals are as follows:

1.	RXDONE	Indicates that a receive operation is complete.
2.	TXDONE	Indicates that a transmit operation is complete.
3.	ACTCHG	Indicates that the RXACTIVE flag has changed state.
4.	SIGCHG	Indicates that one of the modem signals in MODSIGS has changed state.
5.	DLYEND	Indicates that DelayProc may now return, even if the delay period has not expired.

Start-up and Shutdown Considerations

The application must be ready to receive signals when it calls StartSync. Signals may come at any point until control returns from EndSync. Therefore, the application leaves its signal handling mechanism intact until control returns from EndSync.

Special Considerations

Some multi-tasking systems require that all interrupt service routines be installed using a special system call. This gives the multi-tasking system a chance to surround the user's interrupt service routine with its own entry and exit code needed to support multi-tasking operation. Since HSI specifies that the synchronous driver install its own interrupt service routine directly, the special entry/exit code is not executed for interrupts from the synchronous device.

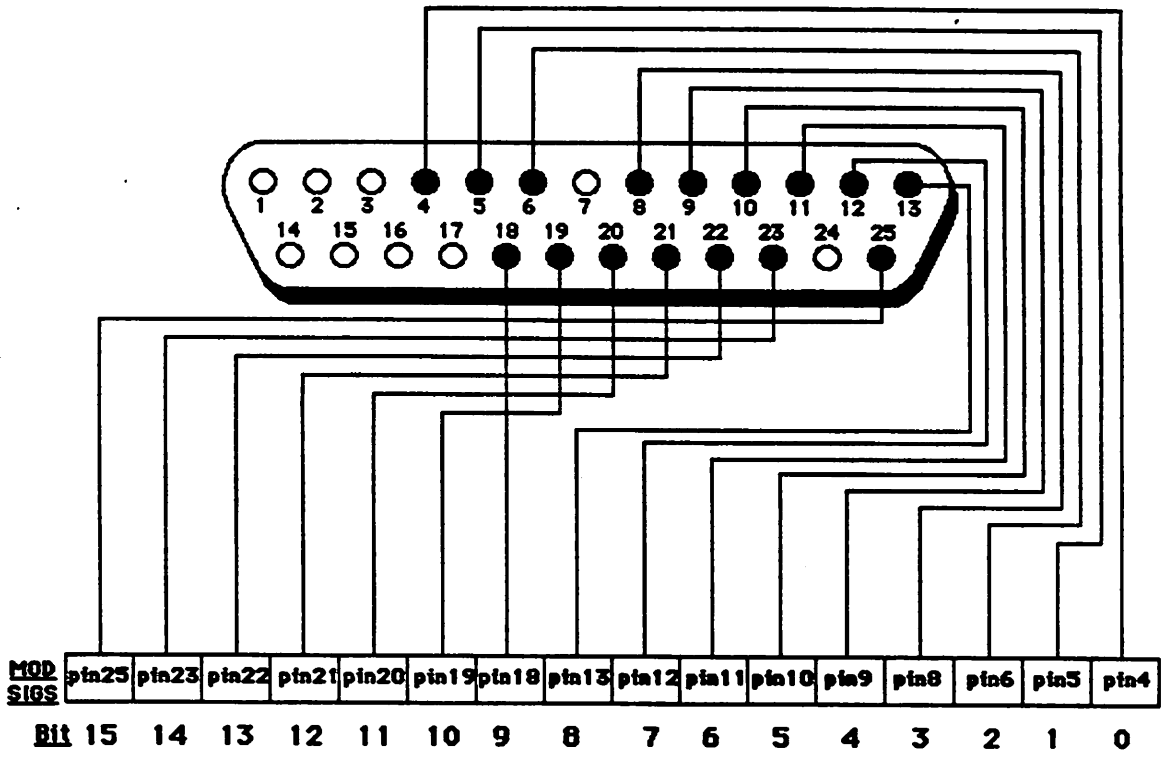
Most interrupts simply transfer one byte of data and return, but problems may arise when interrupts result in a call to SignalProc. If SignalProc makes a system call to "wake up" a task, post an event, etc., the system will believe that this call was made by the interrupted process since the special interrupt entry code was not executed. This is not necessarily a problem; however, it can lead to deep stack nesting if SignalProc calls are made faster than they can be processed.

A solution to this problem is to install a second, dummy interrupt service routine with the aid of the system call, and invoke it from SignalProc by means of a software interrupt mechanism. By making the multi-tasking calls inside this second interrupt service routine, they will be properly preceded and followed by the system's interrupt entry/exit code.

CHAPTER 6--MODEM SIGNALS

Modem Signal Mapping

The InterfaceRec field, MODSIGS, contains the current state of the modem signals, both inputs and outputs. MODSIGS is a 16-bit word, providing for 16 of the 25 possible RS-232 signals.

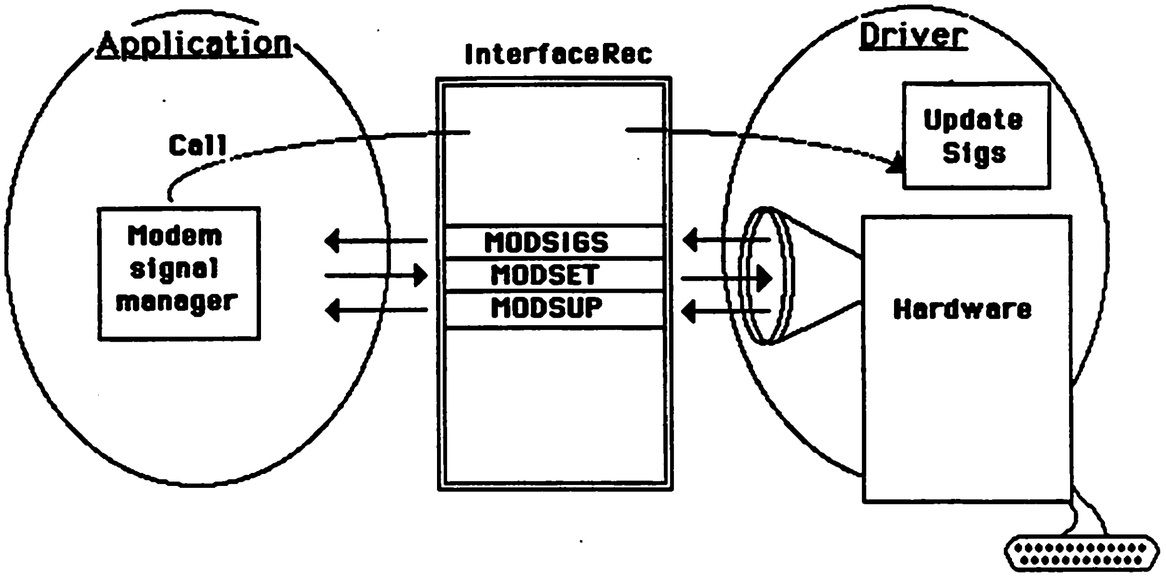


The nine signals which aren't provided for are the data, clock, and ground signals; any signal that the application might want should be there, as long as the signal is actually supported by the hardware.

All of the commonly used modem signals are provided for: RTS (pin 4), CTS (pin 5), DCD (pin 8), DSR (pin 6), DTR (pin 20), Ring Indicate (pin 22),and Rate Select (pin 23). In addition, signals that have been redefined for special purposes by particular vendors are present, such as IBM's Test (pin 18), Test Indicate (pin 25), and Select Standby (pin 11).

Components of Modem Signal Support

The complete modem signal support package consists of three bit-mapped fields in the InterfaceRec: MODSIGS, MODSET, and MODSUP, (plus the driver procedure UpdateSigs). MODSIGS and MODSUP are set by the synchronous driver and read by the application; MODSET is set by the application and read by the driver.

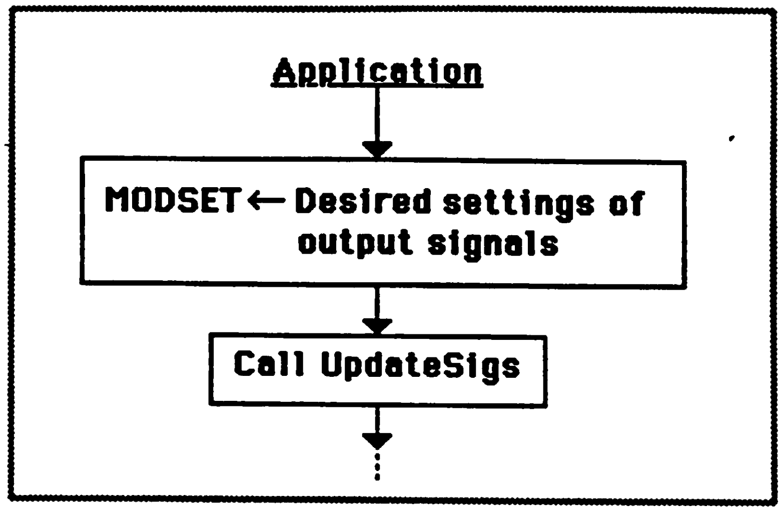


Reading the Modem Signals

The synchronous driver maintains MODSIGS to reflect the current status of the hardware-supported modem signals. Whether a signal functions as an input or an output, the corresponding bit in MODSIGS is a 1 if that signal is asserted, and a 0 if it is not. Any time a bit in MODSIGS changes state, the driver sends the signal, SIGCHG, to the application.

Controlling Output Modem Signals

To control modem signals that function as outputs, the application stores its modem signal settings in MODSET, then calls UpdateSigs.



Inside UpdateSigs the synchronous driver copies the bit settings in MODSET to the actual signal outputs. Bits in MODSET corresponding to input signals are ignored. At the same time, the driver updates MODSIGS to reflect the new settings of the output signals, and sends a SIGCHG if necessary.

Supported vs. Unsupported Modem Signals

Because all hardware devices do not support the same subset of the modem signals, the application must be informed as to which signals are supported in the particular hardware device being used. The InterfaceRec field MODSUP performs this function. As part of the processing performed by Preprocess, the driver sets each bit in MODSUP to one that corresponds to a supported modem signal. A zero bit indicates that the corresponding signal is not supported. The unsupported signals are always reported as zeros in MODSIGS. The driver ignores the bits in MODSET corresponding to unsupported signals.

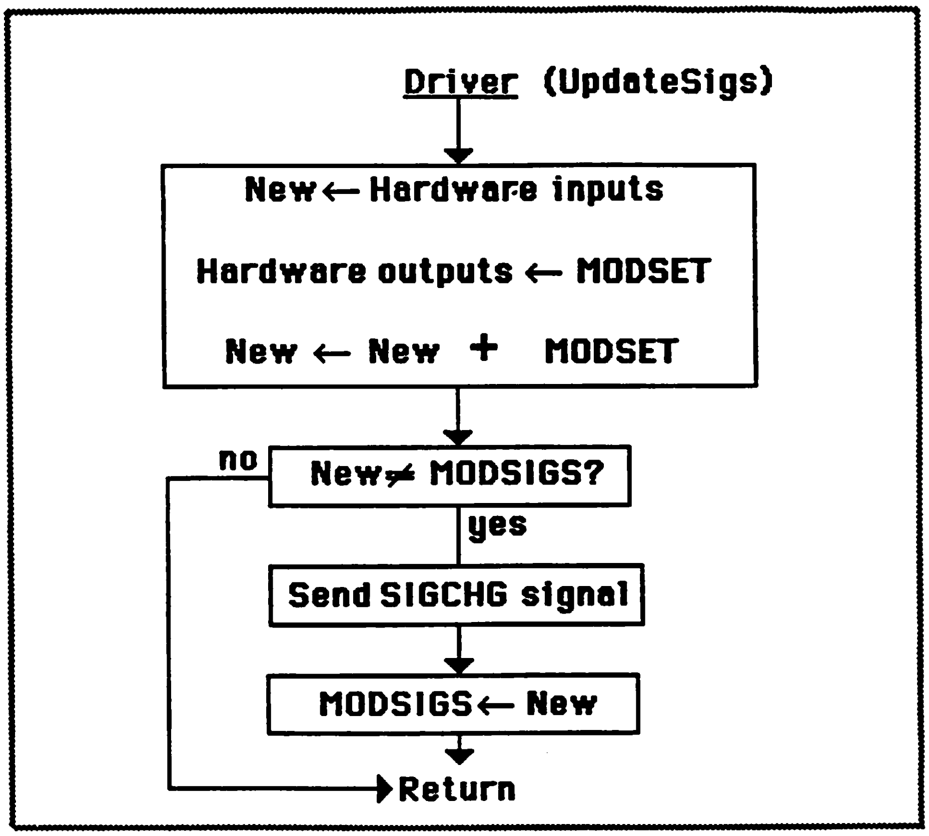
Input vs. Output Modem Signals

HSI does not specify which signals are inputs or outputs, except for RTS, CTS, and DCD. Where these signals are supported, CTS and DCD are always inputs and RTS is always an output. Chapter 2 describes how these three signals are used to regulate transmit and receive operations.

Updating Modem Signals Updated?

Many hardware devices generate interrupts when a modem signal changes state. Thereby, the synchronous driver can automatically track the status of these signals in the MODSIGS field and generate SIGCHG signals as necessary. HSI does not require this automatic updating for all signals, since some hardware devices do not provide interrupts on modem signal changes. To make sure that MODSIGS reflects the current status of all supported modem signals, the application periodically calls UpdateSigs.

UpdateSigs causes the synchronous driver to read the current status of the input signals, copy the settings in MODSET to the output signals, and construct a new MODSIGS word reflecting the current state of all signals. It also issues the SIGCHG signal if the new MODSIGS is different from the previous one.



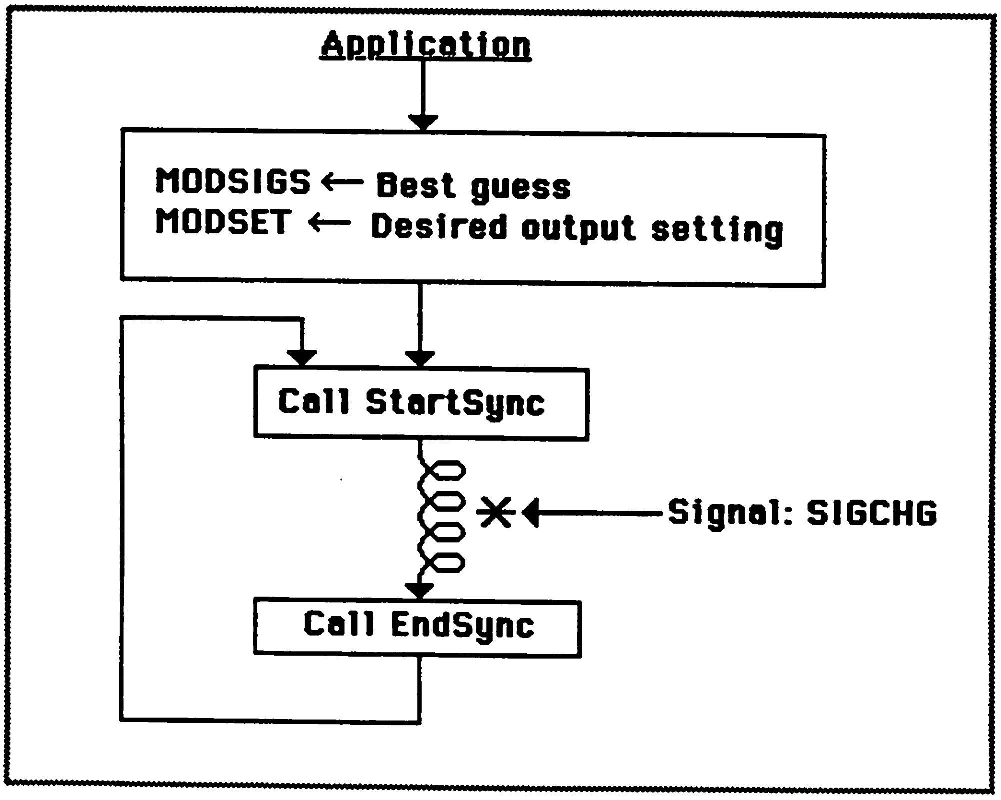
Calling UpdateSigs updates the signals as soon as possible, but not always immediately. Some devices have states in which the modem signal registers are inaccessible. In such cases the modem signals are actually updated some time after control returns from UpdateSigs.

Start-up and Shutdown Considerations

UpdateSigs is automatically called by the synchronous driver when synchronous operation is being started. The driver constructs a new MODSIGS word as usual and, before storing it in the InterfaceRec, compares it to the value of MODSIGS already there, sending a SIGCHG signal if they differ. When the driver terminates, MODSIGS is left the same as its last update. These features allow for smooth transitions into and out of synchronous driver control.

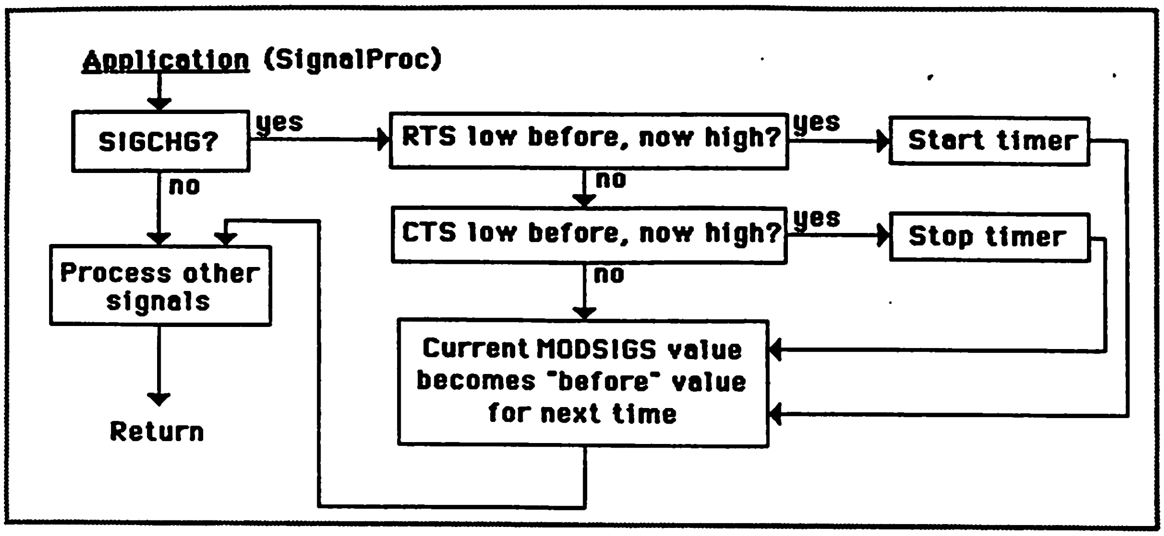
If the application has an estimate of the status of the modem signals, it sets MODSIGS to reflect this estimate before calling StartSync for the first time. It also sets MODSET to reflect the desired settings of the output modem signals. Subsequent calls to StartSync (following EndSync calls) are made without changing MODSIGS, unless the application has additional knowledge of the status of the modem signals from some other source.

When StartSync is called, the driver automatically makes any necessary corrections to MODSIGS to reflect the actual status of the modem signals, informing the application when necessary via the SIGCHG signal.



Programming with Modem Signals

A number of functions can be programmed into the application making use of modem signal support. For example, the delay between the time RTS is raised and CTS goes high can be measured; an error message can be printed if the delay exceeds an established time-out value.


Application (SignalProc)
SIGCHG?
yes
no
RTS low before, now high?
yes
no
Start timer
CTS low before, now high?
yes
no
Stop timer
Process other signals
Return
Current MODSIGS value becomes "before" value for next time

Some devices have on-line/off-line states under the control of DTR; the application can control DTR to place these types of devices in the desired state. Devices may have other states (such as the Link Problem Determination Aid (LPDA) facility in some IBM modems) that the application can invoke by manipulating other modem signals. In all cases, however, the application first checks MODSUP to make sure that the modem signals it will be using are actually supported by the hardware.

CHAPTER 7--TRACE FACILITY

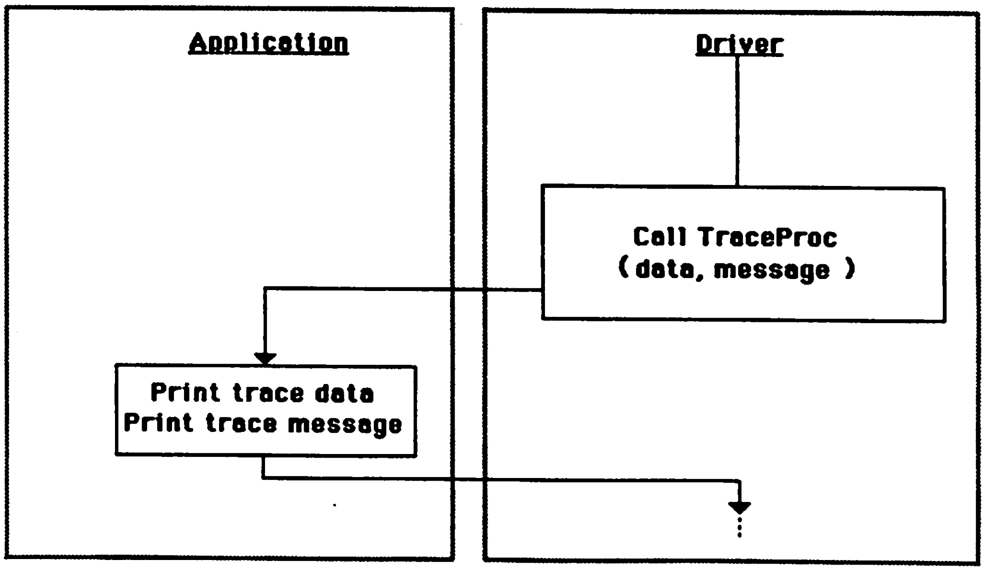
What is the Trace Facility?

HSI defines a simple trace facility that system developers may use to help them debug driver code. The trace facility lets the driver send trace data to the application, which is then responsible for displaying or printing it.

How the Trace Facility is Used

A trace macro is usually defined in the synchronous driver source program, which the programmer may code anywhere to obtain trace data. The trace data may be any 16-bit value, such as the contents of a CPU register or memory location. Paired with the trace data is a 2-character mnemonic code, called the trace message, that identifies the data.

The trace macro generates a call instruction to the TraceProc service routine. The trace data and trace message are passed as parameters to TraceProc and the address of the InterfaceRec. Inside TraceProc, the application prints or displays the trace data and message, returning control to the driver as quickly as possible.



The trace facility is an optional feature of HSI to be used for debug purposes only. All trace macros should be removed from driver code that will be put in a production environment.

APPENDIX A--INTERFACE RECORD LAYOUT

General Information

All numerical values in the InterfaceRec are unsigned binary. Two-byte values and addresses are stored in the customary format for the machine being used. For the IBM PC, two-byte values are stored least-significant byte first; addresses take the form segment:offset and are stored offset first.

In this document, the least significant (right-most) bit is bit 0.

Explanation of Columns

-	Offset	Byte offset of field from beginning of InterfaceRec.
-	Field name	Name this document uses to refer to field.
-	Bytes	Size of field, in bytes.
-	Set by	"A" means application sets field and driver reads it; "D" means driver sets field and application reads it; "D/A" means field is set and read by both driver and application. Note: There may be unusual circumstances, described in this document, in which the "Set by" information provided here is violated.
-	Description	Comments describing use of field.

Section 1--Device Identification

Offset	Field name	Bytes	Set by	Description
0	STRING	100	D/A	Used in conjunction with the RECALL facility of Preprocess, this 100-character buffer is used to store a character string that the synchronous driver wishes to pass to an alternate driver, and to receive the reply string from the alternate driver.
100	DEVQUAL	2	A	Device qualifier used to assist Preprocess in device identification.
102	IOPORT	2	A	For some choices of DEVQUAL, the application must store the device I/O port address here.
104	INTVEC	2	A	For some choices of DEVQUAL, the application must store the device interrupt vector number here.

Section 2--Synchronous Parameter Selection

Offset	Field name	Bytes	Set by	Description
106	BSC	1	A	1 selects BISYNC mode; 0 selects SDLC/HDLC mode.
107	ADDRDET	1	A	1 enables the SDLC/HDLC address detect option; 0 disables this option.
108	SDLCADDR	1	A	If the SDLC/HDLC address detect option is enabled, the desired station address is stored here.
109	BSCSYNC	1	A	In BISYNC mode, the desired SYN character is stored here.
110	NRZI	1	A	1 selects NRZI encoding and decoding; 0 selects NRZ.
111	FLAGIDLE	1	A	1 specifies that the transmitter will send continuous flags (for SDLC/HDLC mode) or SYN characters (for BISYNC mode) when idle; 0 specifies that a continuous mark level will be sent.
112	WRAP	1	A	1 enables the wrap option if it is available; 0 disables this option.
113		9		Reserved for future parameters.

Section 3--Data Exchange Interface

** RECEIVE SIDE **

Offset	Field name	Bytes	Set by	Description
122	RXACTIVE	1	D	1 means the data link is active; 0 means it is idle.
123	RXREADY	1	D/A	Driver sets to 1 when receive operation ends. Application clears to 0 after picking up message.
124	RXINPROG	1	D	Driver sets to 1 when receive operation begins and clears to 0 when operation is completed.
125		1		This byte is not used.
126	RXERROR	2	D	After receive operation, driver stores error code here (or 0 if no error).
128	RXCOUNT	2	D	After receive operation, driver stores number of characters received here.
130	RXINDEX	2	D	Driver uses this field to mark place in buffer for next character.
132	RXBUFSIZ	2	A	Application stores size of receive buffer here.
134	RXBUF	4	A	Application stores address of receive buffer here.
138	RXNXTBUF	4	A	Application stores address of next receive buffer here.
142	RXINTERR	2	D	Driver may store any internal error code here, otherwise this field is 0.
144	RXERRCNT	40	D	Array of 20 error counters for receive side.

Offset	Field name	Bytes	Set by	Description
				** TRANSMIT SIDE **
184	TXREADY	1	D/A	Application sets to 1 when transmit buffer is ready; driver clears to 0 when transmit operation begins.
185	TXINPROG	1	D	Driver sets to 1 when transmit operation begins and clears to 0 when operation is completed.
186	TXERROR	2	D	After transmit operation, driver stores error code here (or 0 if no error).
188	TXINDEX	2	D	Driver uses this field to mark place in buffer to fetch next character.
190	TXCOUNT	2	A	Application stores size of transmit message here.
192	TXBUF	4	A	Application stores address of transmit buffer here.
196	TXINTERR	2	D	Driver may store any internal error code here, otherwise this field is 0.
198	TXERRCNT	40	D	Array of 20 error counters for transmit side.
				** MODEM SIGNALS **
238	MODSIGS	2	D	Current status of all modem signals.
240	MODSET	2	A	Desired state of output modem signals.
242	MODSUP	2	D	Indicates which modem signals are supported by hardware.

Section 4--Service Routine Addresses

Offset	Field name	Bytes	Set by	Description
244	SIGPROC	4	A	Application stores address of SignalProc here.
248	RXCHPROC	4	A	Application stores address of RxCharProc here.
252	TXCHPROC	4	A	Application stores address of TxCharProc here.
256	DLYPROC	4	A	Application stores address of DelayProc here.
260	TRACEPRC	4	A	Application stores address of TraceProc here.

Section 5--Reserved Areas

Offset	Field name	Bytes	Set by	Description
264	APPLAREA	48	A	Reserved for application to use as it wishes.
312	DRIVAREA	48	D	Reserved for driver's internal use.

The total size of the InterfaceRec is 360 bytes.

APPENDIX B--PROCEDURE CALLING CONVENTIONS FOR THE IBM PC ENVIRONMENT

Other Environments

This appendix describes how the driver routines and service routines are called in the IBM PC environment. It explains what parameters are passed, how they are passed, and how the CPU registers are affected. For non-IBM PC environments, the same parameters are passed to each subroutine, but the detailed calling conventions may differ.

General Information

All procedure calls and returns are of the "FAR" variety. All addresses are of the form segment:offset. Data is pushed on the stack in the usual 8086 manner (segment first, etc.). The remark "All registers = garbage" is made in reference to a subroutine call does not apply to registers such as CS, IP, SS, and SP.

The caller is to remove parameters from the stack after control returns from a routine call.

Enabling and Disabling Interrupts

Interrupts may be disabled temporarily inside the driver routines to protect critical code sections. They are not blindly reenabled after the critical section, but rather are restored to the enabled/disabled state prevailing when the driver routine was called.

With the exception of RxCharProc and TxCharProc, which are always called with interrupts disabled, the other service routines are usually called with interrupts enabled. Calls with interrupts disabled would only occur as a consequence of the application's calling a driver routine with interrupts disabled.

Linkage Considerations

The application needs to know the addresses of the five driver routines. HSI does not specify how this is accomplished, but two techniques are suggested:

1. Combine the driver and application source programs; compile/assemble them together and link edit in the usual manner.

2. Provide a table of entry points at the beginning of the driver code segment.

The driver obtains the addresses of the five service routines from the fields defined for this purpose in the InterfaceRec.

Explanation of Headings

Entry:	Conditions that are true before the subroutine is called.
Exit:	Conditions that are true after the subroutine returns.

Driver Routines

1. Preprocess

   Entry: Address of InterfaceRec pushed on stack

   Exit: DS, SI, DI, BP = unchanged
   AX = result code
   BX = result code
   All other registers = garbage

   Result codes: 0 -- OK
   1 -- RECALL
   2 -- BADQUAL
   3 -- BADCALL
   4 -- MODENS
   5 -- WRAPNS
   6 -- METHDNS

2. StartSync

   Entry: Address of InterfaceRec pushed on stack

   Exit: DS, SI, DI, BP = unchanged
   All other registers = garbage

3. EndSync

   Entry: Address of InterfaceRec pushed on stack

   Exit: DS, SI, DI, BP = unchanged
   All other registers = garbage

4. StartTx

   Entry: Address of InterfaceRec pushed on stack

   Exit: DS, SI, DI, BP = unchanged
   All other registers = garbage

5. UpdateSigs

   Entry: Address of InterfaceRec pushed on stack

   Exit: DS, SI, DI, BP = unchanged
   All other registers = garbage

Service Routines

1. SignalProc

   Entry: Address of InterfaceRec pushed on stack (first)
   Signal word pushed on stack (second)
   DS = DS value stored by synchronous driver
   when StartSync last entered.

   Exit: All registers = garbage

   Signal bits: Bit 0 -- RXDONE
   Bit 1 -- TXDONE
   Bit 2 -- ACTCHG
   Bit 3 -- SIGCHG
   Bit 4 -- DLYEND
   Bits 5 through 7 are ignored.

2. RxCharProc

   Entry: DS:SI = address of InterfaceRec
   AL = character just received

   Exit: DS, SI, CX, DX, BP = unchanged
   BH = option bits (set bit to 1 to select that option)
   BL = error code or zero (for ENDMSG option)
   All other registers = garbage

   Option bits: Bit 0 -- IGNORE
   Bit 1 -- ENDMSG
   Bits 2 through 7 are ignored

3. TxCharProc

   Entry: DS:SI = address of InterfaceRec
   AL = character about to be transmitted

Exit: DS, SI, AL, CX, DX, BP = unchanged
BH = option bits (set bit to 1 to select that option)
BL = specified character (for INSERT and SUBST options)
All other registers = garbage

Option bits: Bit 0 -- INSERT
Bit 1 -- SUBST
Bits 2 through 7 are ignored

4. DelayProc

Entry: Address of InterfaceRec pushed on stack (first).
Delay interval in milliseconds pushed on stack (second).
DS = DS value stored by synchronous driver
when StartSync last entered.

Exit: All registers = garbage

5. TraceProc

Entry: Address of InterfaceRec pushed on stack (first).
16-bit trace data word pushed on stack (second).
2-character trace message word pushed on stack (third).
DS = DS value stored by synchronous driver
when StartSync last entered.

Exit: All registers = garbage

APPENDIX C--RESERVED DEVICE QUALIFIERS

The following device qualifiers are reserved for use with HSD:

DEVQUAL = 0.	Indicates that the device has an asynchronous channel at a nonstandard address. The application stores the I/O port address and interrupt vector number in IOPORT and INTVEC, respectively. The driver queries the device for further identification by sending it AT commands via the RECALL facility of Preprocess.
DEVQUAL = 1.	Same as DEVQUAL = 0, but the asynchronous channel is at the standard address for COM1. The driver sets IOPORT to 3F8H and INTVEC to 4.
DEVQUAL = 2.	Same as DEVQUAL = 0, but the asynchronous channel is at the standard address for COM2. The driver sets IOPORT to 2F8H and INTVEC to 3.
DEVQUAL = 3.	Indicates that the device is an IBM SDLC Communications Adapter. The driver sets IOPORT to 388H and INTVEC to 3.

NOTE: Software developers may register additional values of DEVQUAL for inclusion in this Appendix by calling Hayes Customer Service (404-441-1617). This service is extended to licensees of HSD only. Hayes Customer Service personnel are available from 8 a.m. to 8 p.m. EST Monday through Friday.

Hayes®

Hayes Microcomputer Products, Inc.
705 Westech Drive, Norcross, Georgia 30092
Telephone (404) 449-8791 Telex 703500 HAYES USA

mailing address:
P.O. Box 105203, Atlanta, Georgia 30348

Dear Developer:

Thank you for your interest in the Hayes Synchronous Interface (HSI) and the Hayes Synchronous Driver (HSD).

Enclosed is your copy of:

1. Technical References: Hayes Synchronous Interface for Applications Software.
2. Two contracts for licensing the Hayes Synchronous Driver.
3. A return envelope for licensing the Hayes Synchronous Driver.

Should you desire to use the Hayes Synchronous Driver, you will need to sign both copies of the license agreement in order to receive the actual code that constitutes the Hayes Synchronous Driver. Once the agreement is executed, we will send you the Hayes Synchronous Driver technical reference guide with sample source code, and a disk containing the HSD source code.

Please review the agreement carefully, and have an authorized representative of your company execute both copies as follows:

1. Sign at the bottom of the front page.
2. Complete all information required on the back page.
3. Initial the back page.

When the agreements have been completed, please return both copies to us with your $250 license fee in the enclosed return envelope to the attention of the Contracts Administration. Once we have executed the agreement, we will send you the HSD material described above and a signed copy of the agreement.

Thank you for your interest in what we think is an important interface in the PC communications environment. If you have any other questions about this or our Hayes Developer Support Program please call me at (404) 449-8791.

Sincerely,

Kathy Hamdy Swink

Kathy Hamdy-Swink
ISHV Relations Manager

Hayes® Microcomputer Products, Inc.
Software Developer License Agreement

Hayes Microcomputer Products, Inc. ("Hayes") and the undersigned licensee ("Licensee") agree that the following terms and conditions shall govern the license to Licensee by Hayes of the computer software programs designated on Exhibit A, as amended by the parties from time to time (the "Software") and associated documentation (the "Documentation"). For purposes of this Agreement, unless otherwise indicated on Exhibit A, the term "Software" shall mean the object code embodiment of such Software and all enhancements and updates thereof provided hereunder.

1. License. Hayes grants to Licensee a non-exclusive, non-transferable, world wide license under Hayes' copyright to the Software and the Documentation (a) to use, reproduce and distribute copies of the Software in object code form only when incorporated into an application program used internally by Licensee or distributed by Licensee to end users by sale or license subject to the terms and conditions of this Agreement (the "Application"); (b) to the extent that the Software is licensed in source code form, to reproduce and modify such Software for incorporation into the Application; and (c) to use the Documentation internally and, to the extent authorized by Hayes in writing from time to time, to reproduce and distribute to end users copies of designated portions of the Documentation to the extent that the same is incorporated into user documentation for the Application. Licensee may not distribute the Software in source code form, by itself or as marketed by Hayes and may not disassemble or decompile any Software licensed in object code form. During the term of this Agreement, Licensee shall be entitled to receive enhancements and updates of the unmodified Software and the Documentation which may be released by Hayes, in its sole discretion, to all licensees of the Software.

2. Ownership of Software. Hayes owns the Software and the Documentation, the copyrights thereto and any copies thereof licensed hereunder. Hayes has the authority to enter into this Agreement and to grant to Licensee a license to distribute the Software and Documentation. Any modifications of the Software developed by Licensee shall be the property of Licensee, subject to the rights of Hayes under this Agreement and applicable copyright laws.

3. Term. This Agreement shall become effective upon the date accepted by Hayes, shall continue for a period of one year thereafter and shall be automatically renewed for successive one year terms upon payment of the annual license fee as provided in Section 4 and upon execution by Licensee of any modifications to this Agreement which Hayes may specify from time to time, unless earlier terminated in accordance with Section 10 below.

4. License Fee. Upon execution of this Agreement and prior to the commencement of any annual renewal term, Licensee shall pay to Hayes the amount set forth in Exhibit A. Hayes reserves the right to change the License fee for the Software upon thiry (30) days written notice prior to the commencement of any annual renewal term.

5. Licensee Applications. Licensee shall be entitled to incorporate the Software into any Application identified in Exhibit A. Upon prior written notification to Hayes, Licensee may add Applications at no additional charge, and the terms and conditions of this Agreement shall apply to each such addition. For the purpose of enabling Hayes to refer end users to the appropriate persons for assistance, Licensee shall provide Hayes with information regarding available customer support for each Application, including either a general technical description of or a comprehensive sales brochure for, each Application upon Licensee's distribution of the Application to a third party.

6. Labeling. Licensee shall not remove any copyright notices or other proprietary legends contained in or on the original Software, and shall reproduce and include any Hayes copyright notice as it appears in or on the Software on all copies of and packaging for the Applications. Except as set forth in this Section or otherwise permitted by law, Licensee shall have no right to use any of Hayes' trademarks or tradenames without Hayes' specific prior written consent. Licensee shall cause the following warranty disclaimer to be printed conspicuously on the packaging for any Application:

THIS PACKAGE CONTAINS CERTAIN SOFTWARE COPYRIGHTED BY HAYES MICROCOMPUTER PRODUCTS, INC. ("HAYES") AND LICENSED TO LICENSEE. HAYES MAKES NO EXPRESS OR IMPLIED WARRANTIES OF ANY KIND FOR THE SOFTWARE WHICH IS PROVIDED ON AN "AS IS" BASIS, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HAYES SHALL HAVE NO LIABILITY OF ANY KIND TO YOU, INCLUDING ANY LIABILITY FOR SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES.

7. Disclaimer of Warranty. THE SOFTWARE AND THE DOCUMENTATION ARE PROVIDED TO LICENSEE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HAYES SHALL NOT BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, AND THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY LICENSEE, INCLUDING ALL COSTS OF SERVICING, REPAIRS OR CORRECTIONS.

8. Indemnification by Licensee. Except to the extent provided in Section 9 below, Licensee shall indemnify Hayes and hold it harmless from any and all claims, losses, liabilities, expenses (including reasonable attorneys' fees) and damages incurred by Hayes as a result of or arising directly or indirectly from (a) Licensee's use, reproduction or distribution of the Software and the Documentation; or (b) any acts, omissions, misrepresentations, warranties, covenants or obligations hereunder by Licensee or its employees, agents or contractors.

9. Indemnification by Hayes. Hayes shall indemnify Licensee and hold it harmless from any and all claims, losses, liabilities, expenses (including reasonable attorneys' fees) and damages incurred by Licensee, based on a claim that the use or other distribution of the Software as authorized under this Agreement infringes any United States patent, copyright or trade secret of any third party. Licensee shall promptly notify Hayes in writing of any such claim, and provide Hayes with all reasonable assistance in the defense or settlement thereof. In the event of such a claim of infringement, Hayes shall, at its option, (a) attempt to obtain for Licensee a license to continue its use and distribution of the Software; (b) modify or replace the Software with a comparable product which is noninfringing; or (c) refund the license fees previously paid and terminate this Agreement. This Section states Licensee's sole and exclusive remedy for claims of infringement. In no event shall Hayes' liability to Licensee for such claims or for any other claim based on this Agreement exceed the license fees paid by Licensee to Hayes hereunder.

10. Termination. Hayes may terminate this Agreement upon written notice if Licensee breaches any of the terms and conditions of this Agreement and fails to remedy such breach within 30 days after written notification thereof. Termination shall not adversely affect the rights of any third party who is a licensee of any Application at the time of termination. Upon termination, Licensee shall promptly destroy or return to Hayes all copies of the Software and the Documentation in its possession, custody or control; shall discontinue the making of copies of the Software, the Documentation and the Applications and shall immediately cease distribution by sale, lease or otherwise of any copies of the Application then or thereafter in its possession, custody or control.

11. Miscellaneous. This Agreement shall not be construed as creating an agency, partnership or any other form of legal association between the parties. This Agreement constitutes the entire understanding between the parties concerning the subject matter hereof and may not be assigned by Licensee. Any waiver or amendment of any provision of this Agreement shall be effective only if in writing and signed by both parties. Nothing contained in this Agreement shall be construed as conferring upon Licensee by implication, estoppel or otherwise, any license or other right under any patent. This Agreement shall be governed by and construed in accordance with the laws of the state of Georgia.

HAYES MICROCOMPUTER PRODUCTS, INC.

By: ____________________

Title: ____________________

Acceptance Date: ____________________

LICENSEE: ____________________

By: ____________________

Title: ____________________

Date: ____________________

085-1002

Hayes® Microcomputer Products, Inc.

Exhibit A to Software Developer License Agreement

1. Licensee:

Name: ____________________

Address: ____________________

____

____

2. Software Licensed:

HSD

3. Annual License fee:

$250

4. Licensee Contact:

Name: ____________________

Title: ____________________

Phone: ____________________

5. Description(s) of Application(s):

____

____

____

6. Licensee's Commercial software Product(s) Incorporating the Application(s):

____

____

____

Hayes® Microcomputer Products, Inc.
Software Developer License Agreement

Hayes Microcomputer Products, Inc. ("Hayes") and the undersigned licensee ("Licensee") agree that the following terms and conditions shall govern the license to Licensee by Hayes of the computer software programs designated on Exhibit A as amended by the parties from time to time (the "Software") and associated documentation (the "Documentation"). For purposes of this Agreement, unless otherwise indicated on Exhibit A, the term "Software" shall mean the object code embodiment of such Software and all enhancements and updates thereof provided hereunder.

1. License. Hayes grants to Licensee a non-exclusive, non-transferable, world wide license under Hayes' copyright to the Software and the Documentation (a) to use, reproduce and distribute copies of the Software in object code form only when incorporated into an application program used internally by Licensee or distributed by Licensee to end users by sale or license subject to the terms and conditions of this Agreement (the "Application"); (b) to the extent that the Software is licensed in source code form, to reproduce and modify such Software for incorporation into the Application; and (c) to use the Documentation internally and, to the extent authorized by Hayes in writing from time to time, to reproduce and distribute to end users copies of designated portions of the Documentation to the extent that the same is incorporated into user documentation for the Application. Licensee may not distribute the Software in source code form, by itself or as marketed by Hayes and may not disassemble or decompile any Software licensed in object code form. During the term of this Agreement, Licensee shall be entitled to receive enhancements and updates of the unmodified Software and the Documentation which may be released by Hayes, in its sole discretion, to all licensees of the Software.

2. Ownership of Software. Hayes owns the Software and the Documentation, the copyrights thereto and any copies thereof licensed hereunder. Hayes has the authority to enter into this Agreement and to grant to Licensee a license to distribute the Software and Documentation. Any modifications of the Software developed by Licensee shall be the property of Licensee, subject to the rights of Hayes under this Agreement and applicable copyright laws.

3. Term. This Agreement shall become effective upon the date accepted by Hayes, shall continue for a period of one year thereafter and shall be automatically renewed for successive one year terms upon payment of the annual license fee as provided in Section 4 and upon execution by Licensee of any modifications to this Agreement which Hayes may specify from time to time, unless earlier terminated in accordance with Section 10 below.

4. License Fee. Upon execution of this Agreement and prior to the commencement of any annual renewal term, Licensee shall pay to Hayes the amount set forth in Exhibit A. Hayes reserves the right to change the License fee for the Software upon thiry (30) days written notice prior to the commencement of any annual renewal term.

5. Licensee Applications. Licensee shall be entitled to incorporate the Software into any Application identified in Exhibit A. Upon prior written notification to Hayes, Licensee may add Applications at no additional charge, and the terms and conditions of this Agreement shall apply to each such addition. For the purpose of enabling Hayes to refer end users to the appropriate persons for assistance, Licensee shall provide Hayes with information regarding available customer support for each Application, including either a general technical description of or a comprehensive sales brochure for each Application upon Licensee's distribution of the Application to a third party.

6. Labeling. Licensee shall not remove any copyright notices or other proprietary legends contained in or on the original Software, and shall reproduce and include any Hayes copyright notice as it appears in or on the Software on all copies of and packaging for the Applications. Except as set forth in this Section or otherwise permitted by law, Licensee shall have no right to use any of Hayes' trademarks or tradenames without Hayes' specific prior written consent. Licensee shall cause the following warranty disclaimer to be printed conspicuously on the packaging for any Application:

THIS PACKAGE CONTAINS CERTAIN SOFTWARE COPYRIGHTED BY HAYES MICROCOMPUTER PRODUCTS, INC ("HAYES") AND LICENSED TO LICENSEE. HAYES MAKES NO EXPRESS OR IMPLIED WARRANTIES OF ANY KIND FOR THE SOFTWARE WHICH IS PROVIDED ON AN "AS IS" BASIS, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HAYES SHALL HAVE NO LIABILITY OF ANY KIND TO YOU, INCLUDING ANY LIABILITY FOR SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES.

7. Disclaimer of Warranty. THE SOFTWARE AND THE DOCUMENTATION ARE PROVIDED TO LICENSEE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HAYES SHALL NOT BE LIABLE FOR ANY SPECIAL, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES, AND THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY LICENSEE, INCLUDING ALL COSTS OF SERVICING, REPAIRS OR CORRECTIONS.

8. Indemnification by Licensee. Except to the extent provided in Section 9 below, Licensee shall indemnify Hayes and hold it harmless from any and all claims, losses, liabilities, expenses (including reasonable attorneys' fees) and damages incurred by Hayes as a result of or arising directly or indirectly from (a) Licensee's use, reproduction or distribution of the Software and the Documentation; or (b) any acts, omissions, misrepresentations, warranties, covenants or obligations hereunder by Licensee or its employees, agents or contractors.

9. Indemnification by Hayes. Hayes shall indemnify Licensee and hold it harmless from any and all claims, losses, liabilities, expenses (including reasonable attorneys' fees) and damages incurred by Licensee, based on a claim that the use or other distribution of the Software as authorized under this Agreement infringes any United States patent, copyright or trade secret of any third party. Licensee shall promptly notify Hayes in writing of any such claim, and provide Hayes with all reasonable assistance in the defense or settlement thereof. In the event of such a claim of infringement, Hayes shall, at its option, (a) attempt to obtain for Licensee a license to continue its use and distribution of the Software; (b) modify or replace the Software with a comparable product which is noninfringing; or (c) refund the license fees previously paid and terminate this Agreement. This Section states Licensee's sole and exclusive remedy for claims of infringement. In no event shall Hayes' liability to Licensee for such claims or for any other claim based on this Agreement exceed the license fees paid by Licensee to Hayes hereunder.

10. Termination. Hayes may terminate this Agreement upon written notice if Licensee breaches any of the terms and conditions of this Agreement and fails to remedy such breach within 30 days after written notification thereof. Termination shall not adversely affect the rights of any third party who is a licensee of any Application at the time of termination. Upon termination, Licensee shall promptly destroy or return to Hayes all copies of the Software and the Documentation in its possession, custody or control; shall discontinue the making of copies of the Software, the Documentation and the Applications and shall immediately cease distribution by sale, lease or otherwise of any copies of the Application then or thereafter in its possession, custody or control.

11. Miscellaneous. This Agreement shall not be construed as creating an agency, partnership or any other form of legal association between the parties. This Agreement constitutes the entire understanding between the parties concerning the subject matter hereof and may not be assigned by Licensee. Any waiver or amendment of any provision of this Agreement shall be effective only if in writing and signed by both parties. Nothing contained in this Agreement shall be construed as conferring upon Licensee by implication, estoppel or otherwise, any license or other right under any patent. This Agreement shall be governed by and construed in accordance with the laws of the state of Georgia.

HAYES MICROCOMPUTER PRODUCTS, INC.

By: ______________________________

Title: ______________________________

Acceptance Date: ______________________________

LICENSEE: ______________________________

By: ______________________________

Title: ______________________________

Date: ______________________________

085-1002

Hayes® Microcomputer Products, Inc.

Exhibit A to Software Developer License Agreement

1. Licensee:

Name: ____________________

Address: ____________________

____

____

2. Software Licensed:

HSD

3. Annual License fee:

$250

4. Licensee Contact:

Name: ____________________

Title: ____________________

Phone: ____________________

5. Description(s) of Application(s):

____

____

____

6. Licensee's Commercial software Product(s) Incorporating the Application(s):

____

____

____

Hayes®

Hayes Microcomputer Products, Inc.
P.O. Box 105203, Atlanta, Georgia 30348

HAYES MICROCOMPUTER PRODUCTS INC.
P.O. BOX 105203
ATLANTA, GA 30348
Attn: Contracts Administration