Syncdrive

Synchronous Communications Software

User's Manual

QUATECH, INC.
662 Wolf Ledges Parkway
Akron, Ohio 44311

TEL: (330) 434-3154
FAX: (330) 434-1409
www.quatech.com

NOTICE

1 Introduction

Syncdrive is a synchronous communications software driver package designed to aid users of Quatech synchronous communication hardware in the development of their application software. Features of the Syncdrive driver package include:

- Multiple communication channels using up to six MPA-series adapters.
- Support for bit-synchronous (SDLC, HDLC) and byte-synchronous (MONOSYNC, BISYNC) communications.
- DMA capability for bit-synchronous communications.
- User control of most communications parameters.
- Programmer is relieved of the burden of programming the synchronous communications hardware directly.
- User can change or upgrade synchronous communication hardware with minimal modifications to the application software.
- Support for all Quatech MPA-series ISA bus, PCI bus and PCMCIA adapters.
- Versions for DOS, Windows 3.1, Windows 95/98, and OS/2.

Written specifically for use with C, Syncdrive is also compatible with other languages that support large model C type subroutine interfaces.

Syncdrive requires an Intel 80386 processor or later.

1.1 Installation

Installation of Syncdrive varies, depending on the operating system being used. While there is no installation program supplied, the process is quite simple.

1.1.1 DOS

There is no installation procedure for the DOS version of Syncdrive. Syncdrive for DOS is shipped as a function library which is linked directly into the application.

1.1.2 Windows 3.1

Copy the Syncdrive Windows 3.1 DLLs and the Syncdrive VxD to the \windows\system directory. These files can be found in the \WIN31 directory of disk 1 of the Syncdrive diskette set:

SYNC_31.DLL	Syncdrive Windows 3.1 DLL
QTECHDMA.DLL	DMA allocation support DLL
SYNC200.386	Syncdrive Windows 3.1 VxD

Find the [386Enh] section of the system.ini file and add the following line to install the VxD:

device=sync200.386

For the MPA-102 adapter, the VxD file is called SYNC102.386. The line added to system.ini should be:

device=sync102.386

Windows must be restarted to activate the VxD.

1.1.3 Windows 95/98

Copy the Syncdrive Windows 95/98 DLL and the Syncdrive Windows 95/98 VxD to the \windows\system directory on the Windows 95/98 boot drive. These files can be found in the \WIN95 directory of disk 1 of the Syncdrive diskette set:

SYNCDRIV.DLL	Syncdrive Windows 95/98 DLL
SYNC200.VXD	Syncdrive Windows 95/98 VxD

For the MPA-102 adapter, the VxD file is called SYNC102.VXD.

Alternatively, these files may be copied to any directory on the standard search PATH, or they may reside in the same directory where the user's application will reside.

The VxD is dynamically loaded by the DLL as needed. There is no configuration necessary.

1.1.4 OS/2

Install the OS/2 Syncdrive DLL and device driver for the MPA board in any desired directory. We will assume below that the directory "c:\syncdriv" is being used.

These files can be found in the \OS2 directory of disk 1 of the Syncdrive diskette set:

SYNCDRIV.DLL	Syncdrive OS/2 DLL
MPA200.SYS	Syncdrive OS/2 device driver

Find the LIBPATH statement in the config.sys file and add the Syncdrive DLL's directory to it:

LIBPATH=(existing directories);c:\syncdriv

Add a device driver statement at the end of the config.sys:

DEVICE=c:\syncdriv\mpa200.sys

For the MPA-102 adapter, the device driver file is called MPA102.SYS. The line added to config.sys should be:

DEVICE=c:\syncdriv\mpa102.sys

Save config.sys, shutdown OS/2, and reboot. The device driver will display a boot-time message indicating its installation.

2 Synchronous Communications

Most data communications in personal computers is handled asynchronously using standard comm ports. With asynchronous communication, data is transferred one character at a time and with significant overhead due to the addition of start and stop bits required for each character. These additions can decrease the rate of data transfer by 20% or more.

In contrast, synchronous communication transfers data in a format referred to as a "frame". Each frame consists of a block of data plus a fixed amount of overhead from the insertion of control, synchronization, and error detection characters. Since the amount of overhead is independent of the data block size, the percentage of the total transfer time devoted to the frame overhead diminishes as the size of the data block increases.

Synchronous communication is further divided into bit synchronous and byte synchronous transfers. Bit synchronous transfers treat the data block as a series of data bits with no specific character boundaries while byte synchronous transfers treat the data block as a series of fixed length characters.

Syncdrive can transfer data in bit synchronous and byte synchronous modes. The bit synchronous mode may be used to implement such protocols as SDLC or HDLC while the byte synchronous mode may be used to implement protocols such as BISYNC. Syncdrive does not implement any specific protocol itself, but will support most protocols implemented by the application software.

2.1 BIT Synchronous

Syncdrive's bit synchronous mode generates a frame formatted according to Figure 1.

Figure 1 - Syncdrive bit synchronous frame format.

The frame's start flag and end flag characters are used by the hardware for synchronization purposes. When the application transmits a block of data these flag characters are automatically appended to the frame. When receiving a frame, these flag characters are automatically removed from the frame before being returned to the application.

The frame's CRC bytes are used for error detection purposes. When transmitting a block of data the CRC is automatically generated and transmitted after all of the data has been sent. When receiving a frame the CRC is automatically checked to determine if the frame was corrupted. If a CRC error occurs, Syncdrive indicates this error in the *buffer_status* variable of the com_block structure, which is discussed beginning on page 23.

IMPORTANT!

When receiving a frame, the CRC bytes are returned to the application program as part of the data block. The allocated receive buffer must be large enough to hold the received data plus these two additional bytes. The application may ignore the CRC bytes as their validity is determined by Syncdrive**.**

The frame's data area is the buffer provided to Syncdrive by the application. This buffer is the *comm_buffer* array of the com_block structure. When transmitting, the application must allocate a com_block structure and load the data block into the *comm_buffer* array. When receiving, the application must allocate a com_block structure with a large enough *comm_buffer* array to hold the received data PLUS the received CRC.

2.2 BYTE Synchronous

Syncdrive's byte synchronous mode offers the user two different frame formats. Both byte synchronous modes are discussed in the following sections.

Message mode provides the application with a minimal byte synchronous frame format and the hardware calculation and verification of transmit and receive CRCs. Message mode can be used to relieve some of the frame formatting burden from the application.

Block mode uses no frame format or hardware CRC generation. Block mode gives the application full control over the format and generation of the communication frames.

2.3 BYTE Synchronous - message mode

The format of the frame generated by Syncdrive's byte synchronous message mode is shown in Figure 2.

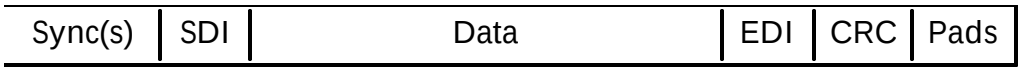

Figure 2 - Syncdrive's Byte Synchronous message mode format.

The frame's initial sync character(s) are used by the hardware for synchronization purposes. When transmitting, the application defines the value to be interpreted as the sync character in the channel configuration structure. Syncdrive automatically adds the number of sync characters specified in the com_block structure to the beginning of the frame. When receiving, the application must define the value(s) to be interpreted as sync characters using sync_command number 5 (see page 76). The sync characters are automatically removed from the frame before the data block is returned to the application.

The frame's SDI, or *start of data indicator*, is a special event character used by Syncdrive to identify the beginning of the data block. When transmitting, the SDI character specified in the com_block structure is automatically added to the frame. When receiving, the application must define the value(s) to be interpreted as an SDI character using sync_command(5). When an SDI character is received, Syncdrive initializes the receiver's CRC and begins a new CRC calculation with the next received character. The SDI character is also returned to the application as part of the received data block.

The frame's EDI, or *end of data indicator*, is a special event character used by Syncdrive to identify the end of the data block. When transmitting, the EDI character specified in the com_block structure is automatically added to the frame. When receiving, the application must define the value(s) to be interpreted as EDI characters using sync_command(5). When an EDI character is received, Syncdrive assumes that the data block is complete and that the next two characters are the frame's CRC. The EDI character is also returned to the application as part of the received data block.

The frame's CRC bytes are used for error detection purposes. When transmitting a block of data, the CRC bytes are automatically generated and transmitted after all of the data has been sent. When receiving a frame the CRC is automatically checked to determine if the frame was corrupted. If a CRC error occurs, Syncdrive indicates this error in the *buffer_status* variable of the com_block structure.

IMPORTANT!

When receiving a frame, the SDI, EDI, and CRC bytes are returned to the application as part of the data block. The allocated receive buffer must be large enough to hold the received data plus these four additional bytes. The application may ignore the CRC bytes since their validity is determined by Syncdrive.

The frame's pad character(s) are used to indicate the end of the frame and place the communication link in the idle state. When transmitting, the application defines the value to be interpreted as the pad character in the channel configuration structure. Syncdrive automatically adds the number of pad characters specified in the com_block structure to the end of the frame. When receiving, the application must define the value(s) to be interpreted as pad characters using sync_command(5). Syncdrive automatically removes all of the pad characters from the frame before the data block is returned to the application.

The frame's data area is the buffer provided to Syncdrive by the application. This buffer is the *comm_buffer* array of the com_block structure discussed beginning on page 23. When transmitting, the application must allocate a com_block structure and load the data block into the *comm_buffer* array. When receiving, the application must allocate a com_block structure with a large enough *comm_buffer* array to hold the received data PLUS the received SDI, EDI, and CRC bytes.

2.4 BYTE Synchronous - block mode

The frame format for Syncdrive's byte synchronous block mode is determined by the application and is treated by Syncdrive as a block of raw unformatted data. The general structure is shown in Figure 3.

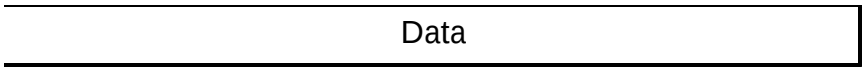

Figure 3 - Syncdrive's byte synchronous block mode format

The byte synchronous block mode frame is the *comm_buffer* provided to Syncdrive as part of the com_block structure discussed beginning on page 23. When transmitting, the application must allocate a com_block structure and load the entire frame, including any sync, pad, and CRC bytes, into the *comm_buffer* array. When receiving, the application must allocate a com_block structure with a large enough *comm_buffer* array to hold the entire received frame, including all sync characters, pad characters, and CRC bytes.

A block mode frame is transparent to Syncdrive. The application is responsible for all data formatting and interpretation. For example, to send a standard BISYNC protocol frame, the application must place at least two sync characters at the start of the buffer. These are followed by a start of data indicator (SDI) of 02H and the actual message being sent. The message is followed by an end of data indicator (EDI) 03H and the CRC. Note, in this mode the CRC must be calculated by the application and then placed in the buffer after the end of data character. Following the CRC, pad characters of FFH could be used.

To receive the same message in block mode, all of the SYNC, SDI, EDI, CRC, and pad characters are placed in the buffer. It is then up to the application to interpret them. Likewise, the CRC of the received data must be calculated and checked against the received CRC (which is also in the buffer following the EDI character).

Actually, however, the SYNC characters preceding the frame do not appear in the buffer because Syncdrive uses the sync character load inhibit feature of the SCC. This feature is automatically turned off when the first non-SYNC character is received.

Independent of the contents of the data block used in block mode, it is recommended that at least two SYNC characters be inserted at the beginning of the data. This will ensure that no data will be lost during receiver synchronization.

3 Syncdrive Data Structures

Syncdrive operations are largely based on three key data structures. These structures are defined in the MPA-X00.H and SYNCDRIV.H include files.

- **channel_cfg** - channel configuration

  This structure defines hardware dependent and protocol dependent information. It includes information on baud rate, interrupts, DMA channels, and protocol information.

- **com_block** - message block and configuration

  This structure defines a particular frame that is being sent or received. It includes information concerning buffer status, control options, and the actual buffer that holds the message data.

- **BufferQueue** - queues of com_block structures

  This structure is used when nonstop processing of frames is desired. It allows multiple com_block structures to be used in one transmit or receive operation.

3.1 The Channel Configuration Structure

The channel configuration structure is used to tell Syncdrive how the application wishes to configure a communication channel. It is allocated by the application. The application must completely initialize the structure before it can be used to configure the channel. A pointer to the channel configuration structure is passed to most Syncdrive API functions in order to identify the channel being operated on.

The channel configuration structure is shown in Figure 4. It is defined in the MPA-X00.H include file.

```
struct channel_cfg
  {
  unsigned char     signature;
  unsigned char     structure_type;
  unsigned short    board_number;
  unsigned char     channel_number;
  unsigned char     operating_mode;
  unsigned char     line_control;
  unsigned char     options;
  unsigned short    base_address;
  unsigned char     tx_dma_channel;
  unsigned char     rx_dma_channel;
  unsigned char     tx_interrupt;
  unsigned char     rx_interrupt;
  unsigned char     crc_mode;
  unsigned char     clock_source;
  unsigned long     clock_rate;
  unsigned long     rx_baud_rate;
  unsigned long     tx_baud_rate;
  unsigned long     read_timeout;
  unsigned long     write_timeout;
  unsigned char     sync_char_1;
  unsigned char     sync_char_2;
  unsigned char     protocol_dependent0;
  unsigned char     protocol_dependent1;
  unsigned char     protocol_dependent2;
  unsigned char     protocol_dependent3;
  unsigned char     protocol_dependent4;
  unsigned char     protocol_dependent5;
  unsigned char     protocol_dependent6;
  unsigned char     protocol_dependent7;
  #if defined(_WINDOWS) && !defined(__WIN95__)
  WIN_HANDLE        (res0, res1)          // Windows 3.1 only
  #endif
  void far *        pt1;
```

Figure 4 - Channel Configuration Structure

3.1.1 signature

Must be set to 0xCC.

3.1.2 structure_type

Choose the appropriate value for the particular MPA-series adapter being used. These values are defined in the SYNCDRIV.H include file.

The MPAC-100 PCI adapter is functionally equivalent to the MPAP-100 PCMCIA adapter. Use MPAP100_ID for the MPAC-100.

MPA100_ID
MPA200_ID
MPA102_ID
MPAP100_ID
MPAP200_ID

3.1.3 board_number

Every MPA-series adapter installed in the system must have a unique *board_number*. Valid values range from 0 to 5.

3.1.4 channel_number

This variable is set to 0 for channel A of a particular board (designated by a unique *board_number)*, or to 1 for channel B. Only the MPA-102 has a channel B. This variable must be set to 0 for all other MPA-series adapters.

3.1.5 operating_mode

Specifies the operating mode of the SCC.

D7	D6	D5	D4	D3	D2	D1	D0	Mode
0	0	-	-	-	-	-	-	Reserved
-	-	0	0	-	-	-	-	NRZ Data Modulation
-	-	0	1	-	-	-	-	NRZI Data Modulation[1]
-	-	1	0	-	-	-	-	FM1 Data Modulation[2]
-	-	1	1	-	-	-	-	FM0 Data Modulation[2]
-	-	-	-	0	0	1	0	MONOSYNC w/8-bit sync
-	-	-	-	0	1	0	0	BISYNC w/16 bit sync
-	-	-	-	1	0	0	0	Bit Sync w/8-bit flags

[1] The *clock_source* field must be set to X32 Clock Source and RX CLK SRC = DPLL for NRZI data modulation (Value = 10011111B).
[2] The *clock_source* field must be set to X16 Clock Source and RX CLK SRC = DPLL for FM1 and FM0 data modulation (Value = 01011111B).

3.1.6 line_control

Defines the data format of the serial data.

D7	D6	D5	D4	D3	D2	D1	D0	Data format
0	0	-	-	-	-	-	-	5 bit transmit data
0	1	-	-	-	-	-	-	7 bit transmit data
1	0	-	-	-	-	-	-	6 bit transmit data
1	1	-	-	-	-	-	-	8 bit transmit data
-	-	0	0	-	-	-	-	5 bit receive data
-	-	0	1	-	-	-	-	7 bit receive data
-	-	1	0	-	-	-	-	6 bit receive data
-	-	1	1	-	-	-	-	8 bit receive data
-	-	-	-	0	0	-	-	Reserved
-	-	-	-	-	-	0	0	No Parity
-	-	-	-	-	-	0	1	Odd Parity
-	-	-	-	-	-	1	1	Even Parity

3.1.7 options

Defines other communications options.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
1	-	-	-	-	-	-	-	Auto Enables
-	1	-	-	-	-	-	-	Address Enable
-	-	1	-	-	-	-	-	Auto Echo Enable
-	-	-	1	-	-	-	-	Loop Back Enable
-	-	-	-	0	0	0	0	Reserved

Auto Enables --- when set, the SCC's CTS input becomes a transmitter enable and its DCD input becomes a receiver enable.

Address Enable --- when set, and if in SDLC (bit sync) mode, the hardware will ignore any frames with SDLC secondary addresses not matching the value programmed into the *sync_char_1* field.

Auto Echo Enable --- when set, TxD and RxD are internally connected, plus the receiver continues to listen to the external RxD pin. Transmitted data is not seen inside or outside the SCC, and CTS is ignored as a transmit enable. (This option has little use for most applications.)

Loop Back Enable --- when set, local loopback is enabled. TxD and RxD are internally connected, and transmitted data is also routed to the external TxD pin. The CTS and DCD signals are ignored as transmit and receive enables. (This option has little use for most applications.)

3.1.8 base_address

The base I/O address of the communication hardware. This value must correspond to the switch setting on the MPA-series adapter.

3.1.9 tx_dma_channel

The DMA channel to be used for bit synchronous transmission. This variable must be set to correspond to the jumper setting on the MPA-series adapter. It should be set to 0 if DMA is not being used for transmit.

DMA cannot be used for byte-synchronous transmission. This variable must be set to 0 if byte-synchronous transmission is selected.

Because the PCMCIA bus does not support DMA, this variable must be set to 0 when a PCMCIA MPA-series adapter is used.

If communications are full duplex, different DMA channels must be chosen for transmit and receive.

3.1.10 rx_dma_channel

The DMA channel to be used for bit synchronous reception. This variable must be set to correspond to the jumper setting on the MPA-series adapter. It should be set to 0 if DMA is not being used for receive.

DMA cannot be used for byte-synchronous reception. This variable must be set to 0 if byte-synchronous reception is selected.

Because the PCMCIA bus does not support DMA, this variable must be set to 0 when a PCMCIA MPA-series adapter is used.

If communications are full duplex, different DMA channels must be chosen for transmit and receive.

3.1.11 tx_interrupt, rx_interrupt

The IRQ level to be used for transmitting and receiving. These variables must be set to correspond to the jumper setting on the MPA-series adapter. For PCMCIA MPA-series adapters, the setting must correspond to the value set by client driver.

MPA-series adapters use only one IRQ. Therefore, the *tx_interrupt* and *rx_interrupt* variables must both be set to the same value.

3.1.12 crc_mode

Specifies the type of CRC calculation to be performed. Typically, CRC-16 initialized to all 0's is used for byte synchronous formats and CCITT CRC initialized to all 1's is used for bit synchronous formats.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
0	-	-	-	-	-	-	-	Initialize CRC to all 0's
1	-	-	-	-	-	-	-	Initialize CRC to all 1's
-	0	0	0	0	0	-	-	Reserved
-	-	-	-	-	-	0	0	No CRC
-	-	-	-	-	-	0	1	CRC-16
-	-	-	-	-	-	1	0	CCITT CRC (SDLC)

3.1.13 clock_source

Sets the source of the transmit and receive data clocks. Bits 6-7 select the rate of the clock relative to the data rate. The clock mode for most applications will be x1 unless NRZI or FM data encoding is used (see discussion of *operating_mode* on page 14).

Bits 3-5 select the source of the received data clock. The RTxC and TRxC pins can be used to receive this clock from the connector. The receive data clock can be self-sourced from the SCC's baud rate generator (BRG) or the DPLL. If clock signals are not present on the connecting cable, or if they are to be ignored, the BRG option is generally suitable and easy to use.

Bits 0-2 select the source of the transmit data clock. The RTxC and TRxC pins can be used to receive this clock from the connector. The transmit data clock can be self-sourced from the SCC's baud rate generator (BRG) or the DPLL. If clock signals are not present on the connecting cable, or if they are to be ignored, the BRG option is generally suitable and easy to use.

Available clock options differ amongst the various models of MPA-series adapters. The options are explained beginning on page 95.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
0	0	-	-	-	-	-	-	x1 Clock Mode
0	1	-	-	-	-	-	-	x16 Clock Mode
1	0	-	-	-	-	-	-	x32 Clock Mode
1	1	-	-	-	-	-	-	x64 Clock Mode
-	-	1	-	-	-	-	-	Output Rx clock
-	-	0	-	-	-	-	-	Input Rx clock
-	-	-	0	0	-	-	-	Rx clock source = RTxC pin
-	-	-	0	1	-	-	-	Rx clock source = TRxC pin
-	-	-	1	0	-	-	-	Rx clock source = BRG
-	-	-	1	1	-	-	-	Rx clock source = DPLL
-	-	-	-	-	1	-	-	Output Tx clock
-	-	-	-	-	0	-	-	Input Tx clock
-	-	-	-	-	-	0	0	Tx clock source = RTxC pin
-	-	-	-	-	-	0	1	Tx clock source = TRxC pin
-	-	-	-	-	-	1	0	Tx clock source = BRG
-	-	-	-	-	-	1	1	Tx clock source = DPLL

The digital phase locked loop (DPLL) feature can be used in the FM0, FM1, and NRZI modes. To enable the DPLL set *operating_mode* (see page 14) to the desired modulation method. The Rx clock source should be set to select the DPLL. Also, the following rules governing the clock oversampling rate with the chosen modulation method must be followed:

Data Encoding	Clock Mode
NRZ	x1
NRZI	x32
FM0	x16
FM1	x16

Syncdrive always uses the BRG input for the DPLL, so the baud rate must be set for x16 or x32 the actual data rate.

For example, to run the receive to demodulate bit synchronous FM0 modulated data, use the DPLL as the clock source the with the clock oversampled in x16 mode. The following variables could be set:

```
operating_mode = 0x38;      // FM0 mode
                            // Bit Sync w/8 bit address

clock_source = 0x5F;        // x16 clock,
                            // input Rx clock
                            // Rx clock source = DPLL output
                            // output Tx clock
                            // Tx clock = DPLL output
```

3.1.14 clock_rate

The frequency of the clock oscillator on the MPA-series adapter. Syncdrive uses this value in its baud rate divisor calculations. The factory default clock rate is 9.8304 MHz, or 9830400. Other clock rates are available upon request. The value of this variable must match the frequency of the clock oscillator!

3.1.15 rx_baud_rate, tx_baud_rate

The desired baud rate of the receiver and transmitter. If the desired baud rate does not allow for an integer baud rate divisor,

Syncdrive will program the hardware for the achievable baud rate closest to that desired.

Each channel runs using a single baud rate generator. The *rx_baud_rate* and *tx_baud_rate* variables should be set to the same value. On single-channel MPA-series adapters, the baud rate generators of both the "A" and "B" channels of the SCC are set to this value.

3.1.16 read_timeout

For Syncdrive for OS/2, this is the number of milliseconds to wait for a frame to be received. The calling thread is blocked during this interval. If this value is 0, a sync_receive call will return immediately, just as it always does under DOS. If this value is -1, the timeout is infinite.

If not under OS/2, this variable should be set to 0.

3.1.17 write_timeout

For Syncdrive for OS/2, this is the number of milliseconds to wait for a frame to be transmitted. The calling thread is blocked during this interval. If this value is 0, a sync_transmit call will return immediately, just as it always does under DOS. If this value is -1, the timeout is infinite.

If not under OS/2, this variable should be set to 0.

3.1.18 sync_char_1

The Tx sync character for MONOSYNC operation, the lower 8 bits of the Tx/Rx sync character for BISYNC operation, or the secondary address for bit sync operation.

- MONOSYNC - 8 bit Tx sync character
- BISYNC - 8 LSBs of 16 bit Tx/Rx sync character
- bit sync - secondary address

3.1.19 sync_char_2

The Rx sync character for MONOSYNC operation, or the upper 8 bits of the Tx/Rx sync character for BISYNC operation. For bit sync operation this variable must be set to the flag character of 0x7E.

- MONOSYNC - 8 bit Rx sync character
- BISYNC - 8 MSBs of Tx/Rx sync character
- bit sync - must be 0x7E (flag character)

3.1.20 protocol_dependent

These variables represent one of several things, depending on which mode the application is using.

protocol_dependent0

BISYNC: Pad character - This is the character that is appended to the end of the BISYNC frame when in message mode.

protocol_dependent1

BISYNC: 0 = Include STX in the CRC calculation.
1 = Do NOT include the STX in the CRC calculation. (Standard BISYNC)

protocol_dependent2

HDLC: 0 = Idle line with syncs (flags)
1 = Idle line with marks (all 1s)

protocol_dependent3 (reserved)

protocol_dependent4 (reserved)

protocol_dependent5 (reserved)

protocol_dependent6 (reserved)

protocol_dependent7 (reserved)

3.1.21 res0, res1, ptr1, ptr1a

Fields used internally by Syncdrive. These fields must not be modified or referenced by the application program.

3.2 The com_block Structure

The com_block structure is used to transfer data blocks between the application and Syncdrive's transmit and receive routines. A com_block structure is allocated by the application by either a simple declaration as a global variable or by a call to the C runtime library malloc function. Using DMA requires the application to allocate com_block structures with the sync_alloc_dma_buffer function.

Four protocol-specific versions and two generic versions of the com_block structure are overlaid using a union declaration. This permits Syncdrive to use the generic forms internally while the application uses a version more suited to the protocol being used. The union declaration in the SYNCDRIV.H include file is shown in Figure 5.

```
union com_block
      {
      struct  MsgBuf                A;              // Generic, short
names.
      struct  GenMsgBuf       B;              // Generic, long names.
      struct  BitTMsgBuf      BitT;           // Bit synchronous transmit.
      struct  BitRMsgBuf      BitR;           // Bit synchronous receive.
      struct  ByteTMsgBuf     ByteT;          // Byte synchronous transmit.
      struct  ByteRMsgBuf     ByteR;          // Byte synchronous receive
```

Figure 5 - The com_block union

IMPORTANT!

The application may inspect a com_block structure at any time but MUST not modify its contents after transmit or receive operations have been started.

3.3 BIT SYNCHRONOUS TRANSMIT com_block

The com_block structure definition for the bit synchronous transmit mode is shown in Figure 6.

```
struct BitTMsgBuf
  {
  #if defined(_WINDOWS) && !defined(__WIN95__)
      WIN_HANDLE (BufferHNDL, res0)       // Windows 3.1 only
  #endif
  unsigned int        buffer_length;
  unsigned int        buffer_pointer;
  unsigned char       buffer_status;
  unsigned char       buffer_error;
  unsigned char       idle_flag;
  unsigned char       reserved_char_1;
  unsigned char       reserved_char_2;
  unsigned char       reserved_char_3;
  unsigned char       reserved_char_4;
  unsigned char       reserved_char_5;
  unsigned char       reserved_char_6;
  unsigned char       dma_page;
  unsigned int        dma_offset;
  unsigned char       comm_buffer[buffer_size];
  };
```

Figure 6 - Bit sync Tx com_block structure

3.3.1 BufferHNDL (Windows 3.1 only)

Present only in the Windows 3.1 version of Syncdrive for backward compatibility with versions of Syncdrive prior to release 4.00. Newly written Windows 3.1 applications should treat this field as reserved.

3.3.2 buffer_length

The number of bytes in the data block. The application must initialize *buffer_length* to the length of the data block contained in *comm_buffer*. The application may inspect *buffer_length* at any time but MUST NOT modify its contents while the transmission is in progress.

3.3.3 buffer_pointer

Byte offset into the *comm_buffer* array of this structure. This variable gets initialized to zero when a transmit operation begins. Syncdrive updates *buffer_pointer* as each character is transmitted. If DMA is being used, this variable is not updated. The application may inspect *buffer_pointer* at any time but MUST NOT modify its contents while the transmission is in progress.

3.3.4 buffer_status

Current status of the communication buffer. The application must set this to zero before starting the transmission. The application software may inspect *buffer_status* at any time but MUST NOT modify its contents after a transmission has started.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
1	-	-	-	-	-	-	-	Done
-	1	-	-	-	-	-	-	Transmitting Data
-	-	1	-	-	-	-	-	Transmitting CRC
-	-	-	-	1	-	-	-	Transmitted Abort
-	-	-	-	-	1	-	-	Error
-	-	-	-	-	-	-	1	Frame Timeout (OS/2)

3.3.5 buffer_error

Indicates an error condition has arisen with respect to Syncdrive's handling of the com_block structure. The application must set this field to zero before starting the transmission, and should not attempt to modify its contents after the transmission has started.

If the TX_RETRANSMITTED bit (0x01) is set, this com_block was retransmitted when a ring mode transmit buffer queue overran and the overrun was permitted. See page 51 for details.

3.3.6 idle_flag

Specifies which state the transmit line should assume after completing the data transfer. If *idle_flag* is 0, the transmitter will keep the line active by continuously sending sync or flag characters after the data transfer is complete. If *idle_flag* is any non-zero, the transmitter will be disabled after transmission, causing the line to assume an idle state.

3.3.7 reserved_char_x

Reserved for future use. All reserved_char fields must be set to 0.

3.3.8 dma_page, dma_offset

Used internally by Syncdrive. These variables must not be changed by the application.

3.3.9 comm_buffer[ ]

Unsigned character array containing the data to be transmitted. The size of *comm_buffer* is user defined with the restriction that the entire com_block structure must reside in a single 64 Kbyte segment. The size of *comm_buffer* is configured at compile time by the buffer_size define statement (default 1000 bytes) found in the SYNCDRIV.H include file.

3.4 BIT SYNCHRONOUS RECEIVE com_block

The com_block structure definition for the bit synchronous receive mode is shown in Figure 7.

```
struct BitRMsgBuf
  {
  #if defined(_WINDOWS) && !defined(__WIN95__)
      WIN_HANDLE (BufferHNDL, res0)       // Windows 3.1 only
  #endif
  unsigned int        buffer_length;
  unsigned int        buffer_pointer;
  unsigned char       buffer_status;
  unsigned char       buffer_error;
  unsigned char       reserved_char_1;
  unsigned char       reserved_char_2;
  unsigned char       reserved_char_3;
  unsigned char       reserved_char_4;
  unsigned char       reserved_char_5;
  unsigned char       reserved_char_6;
  unsigned char       reserved_char_7;
  unsigned char       dma_page;
  unsigned int        dma_offset;
  unsigned char       comm_buffer[buffer_size];
  };
```

Figure 7 - Bit sync Rx com_block structure

3.4.1 BufferHNDL (Windows 3.1 only)

Present only in the Windows 3.1 version of Syncdrive for backward compatibility with versions of Syncdrive prior to release 4.00. Newly written Windows 3.1 applications should treat this field as reserved.

3.4.2 buffer_length

The number of bytes in the receive buffer. The application must initialize *buffer_length* to the length of the available receive buffer. The application may inspect *buffer_length* at any time but MUST NOT modify its contents while the reception is in progress.

3.4.3 buffer_pointer

Byte offset into the *comm_buffer* array of this structure. This variable gets initialized to zero when a receive operation begins. Syncdrive updates *buffer_pointer* as each character is received. If DMA is being used, this variable is updated at the end of the message after all frame characters have been received. The application may inspect *buffer_pointer* at any time but MUST NOT modify its contents while the reception is in progress.

3.4.4 buffer_status

Current status of the communication buffer. The application must set this variable to zero before starting the reception. The application may inspect *buffer_status* at any time but MUST NOT modify its contents after the reception has started.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
1	-	-	-	-	-	-	-	Done
-	1	-	-	-	-	-	-	Receiving Data
-	-	1	-	-	-	-	-	Receiving CRC
-	-	-	1	-	-	-	-	Receive Buffer Full
-	-	-	-	1	-	-	-	Received Abort
-	-	-	-	-	1	-	-	Error (CRC or Rx overrun)
-	-	-	-	-	-	-	1	Frame Timeout (OS/2)

3.4.5 buffer_error

Indicates an error condition has arisen with respect to Syncdrive's handling of the com_block structure. The application must set this field to zero before starting the transmission, and should not attempt to modify its contents after the transmission has started.

If the RX_OVERWRITTEN bit (0x01) is set, this com_block was overwritten when a ring mode receive buffer queue overran and the overrun was permitted. See page 51 for details.

3.4.6 reserved_char_x

Reserved for future use. All reserved_char fields must be set to 0.

3.4.7 dma_page, dma_offset

Used internally by Syncdrive. These variables must not be changed by the application.

3.4.8 comm_buffer[ ]

Unsigned character array used to hold the received data. The size of *comm_buffer* is user defined with the restriction that the entire com_block structure must reside in a single 64 Kbyte segment. The size of *comm_buffer* is configured at compile time by the buffer_size define statement (default 1000 bytes) found in the SYNCDRIV.H include file.

3.5 BYTE SYNCHRONOUS TRANSMIT com_block

The com_block structure definition for the byte synchronous transmit mode is shown in Figure 8.

```
struct ByteTMsgBuf
  {
  #if defined(_WINDOWS) && !defined(__WIN95__)
      WIN_HANDLE (BufferHNDL, res0)       // Windows 3.1 only
  #endif
  unsigned int        buffer_length;
  unsigned int        buffer_pointer;
  unsigned char       buffer_status;
  unsigned char       buffer_error;
  unsigned char       idle_flag;
  unsigned char        misc_options;
  unsigned char       reserved_char_1;
  unsigned char       message_type;
  unsigned char       message_sdi;
  unsigned char       message_edi;
  unsigned char       num_syncs;
  unsigned char       sync_count;
  unsigned char       num_txpads;
  unsigned char       txpad_count;
  unsigned char       comm_buffer[buffer_size];
  };
```

Figure 8 - Byte sync Tx com_block structure

3.5.1 BufferHNDL (Windows 3.1 only)

Present only in the Windows 3.1 version of Syncdrive for backward compatibility with versions of Syncdrive prior to release 4.00. Newly written Windows 3.1 applications should treat this field as reserved.

3.5.2 buffer_length

The number of bytes in the data block. The application must initialize *buffer_length* to the length of the data block contained in *comm_buffer*. The application may inspect *buffer_length* at any time but MUST NOT modify its contents while the transmission is in progress.

3.5.3 buffer_pointer

Byte offset into the *comm_buffer* array of this structure. This variable gets initialized to zero when a transmit operation begins. Syncdrive updates *buffer_pointer* as each character is transmitted. The application may inspect *buffer_pointer* at any time but MUST NOT modify its contents while the transmission is in progress.

3.5.4 buffer_status

Current status of the communication buffer. The application must set this variable to zero before starting the transmission. The application may inspect *buffer_status* at any time but MUST NOT modify its contents after a transmission has started.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
1	-	-	-	-	-	-	-	Done
-	1	-	-	-	-	-	-	Transmitting Data
-	-	1	-	-	-	-	-	Transmitting CRC
-	-	-	-	1	-	-	-	Transmitted Abort
-	-	-	-	-	1	-	-	Error
-	-	-	-	-	-	-	1	Frame Timeout (OS/2)

3.5.5 buffer_error

Indicates an error condition has arisen with respect to Syncdrive's handling of the com_block structure. The application must set this field to zero before starting the transmission, and should not attempt to modify its contents after the transmission has started.

If the TX_RETRANSMITTED bit (0x01) is set, this com_block was retransmitted when a ring mode transmit buffer queue overran and the overrun was permitted. See page 51 for details.

3.5.6 idle_flag

Specifies which state the transmit line should assume after completing the data transfer. If *idle_flag* is 0, the transmitter will keep the line active by continuously sending sync characters after the data transfer is complete. If *idle_flag* is any non-zero value, the transmitter will be disabled after transmission, causing the line to assume an idle state.

3.5.7 misc_options

If bit 0 of this field is set to 1, the block mode byte streaming feature is enabled. When the end of the *comm_buffer* is reached, *buffer_pointer* is reset to 0 and the data in the buffer is transmitted again. Transmission of the frame can be stopped by calling sync_abort_message. The *message_type* field must be set for block mode for this feature to be enabled. All other bits in this field are reserved for future use and should be set to 0.

3.5.8 message_type

Specifies the type of frame to be transmitted. If *message_type* is 0, the data block will be transmitted using byte synchronous message mode. If *message_type* is any non-zero value, the data block will be transmitted using byte synchronous block mode.

3.5.9 message_sdi

Defines the SDI byte to be used for this message. If block mode is specified in the *message_type* field, *message_sdi* is ignored.

3.5.10 message_edi

Defines the EDI byte to be used for this message. If block mode is specified in the *message_type* field, *message_edi* is ignored.

3.5.11 num_syncs

Specifies the number of sync characters to be transmitted before the SDI byte is sent. The total number of syncs will be *(num_syncs* + 2). On block mode transmit two sync characters will be sent out ahead of the block.

3.5.12 sync_count

Used by Syncdrive to record the number of sync characters already sent. The application may inspect *sync_count* at any time but MUST NOT modify its contents after a transmission has started.

3.5.13 num_txpads

Indicates the number of pad characters with which to terminate a message mode frame. This minimum value for this variable is 1.

3.5.14 txpad_count

Used by Syncdrive to record the number of pad characters already sent. The application may inspect *txpad_count* at any time but MUST NOT modify its contents after a transmission has started.

3.5.15 reserved_char_x

Reserved for future use. All reserved_char fields must be set to 0.

3.5.16 comm_buffer[ ]

Unsigned character array used to hold the transmitted data. The size of *comm_buffer* is user defined with the restriction that the entire com_block structure must reside in a single 64 Kbyte segment. The size of *comm_buffer* is configured at compile time by the buffer_size define statement (default 1000 bytes) found in the SYNCDRIV.H include file.

3.6 BYTE SYNCHRONOUS RECEIVE com_block

The com_block structure definition for the byte synchronous receive mode is shown in Figure 9.

```
struct ByteRMsgBuf
  {
  #if defined(_WINDOWS) && !defined(__WIN95__)
      WIN_HANDLE (BufferHNDL, res0)       // Windows 3.1 backward compat.
  #endif
  unsigned int        buffer_length;
  unsigned int        buffer_pointer;
  unsigned char       buffer_status;
  unsigned char       buffer_error;
  unsigned char       reserved_char_1;
  unsigned char       misc_options;
  unsigned char       reserved_char_2;
  unsigned char       message_type;
  unsigned char       reserved_char_3;
  unsigned char       reserved_char_4;
  unsigned char       num_pads;
  unsigned char       pad_count;
  unsigned char       block_term;
  unsigned char       reserved_char_5;
  unsigned char       comm_buffer[buffer_size];
  };
```

Figure 9 - Byte sync Rx com_block structure

3.6.1 BufferHNDL (Windows 3.1 only)

Present only in the Windows 3.1 version of Syncdrive for backward compatibility with versions of Syncdrive prior to release 4.00. Newly written Windows 3.1 applications should treat this field as reserved.

3.6.2 buffer_length

The number of bytes in the receive buffer. The application must initialize *buffer_length* to the length of the available receive buffer. The application may inspect *buffer_length* at any time but MUST NOT modify its contents while the reception is in progress.

3.6.3 buffer_pointer

Byte offset into the *comm_buffer* array of this structure. This variable gets initialized to zero when a receive operation begins. Syncdrive updates *buffer_pointer* as each character is received. The

application program may inspect *buffer_pointer* at any time but MUST NOT modify its contents while the reception is in progress.

3.6.4 buffer_status

Current status of the communication buffer. The application must set this variable to zero before starting the reception. The application may inspect *buffer_status* at any time but MUST NOT modify its contents after the reception has started.

During receive operations, the CRC error bit is set while receiving and may be cleared, depending on whether CRC checking was done on the message and whether the message correctly received. Thus if a BISYNC message was received that does not include a CRC (indicated by the lack of a ETX character in the message), the Error bit will remain set and the Receiving CRC bit will remain cleared. If the CRC was checked, the Receiving CRC bit will be set and then the Error status can be used to indicate a good / bad status on the message.

D7	D6	D5	D4	D3	D2	D1	D0	Status Information
1	-	-	-	-	-	-	-	Done
-	1	-	-	-	-	-	-	Receiving Data
-	-	1	-	-	-	-	-	Receiving CRC
-	-	-	1	-	-	-	-	Receive Buffer Full
-	-	-	-	1	-	-	-	Received Abort
-	-	-	-	-	1	-	-	Error (Rx overrun)
-	-	-	-	-	-	-	1	Frame Timeout (OS/2)

3.6.5 buffer_error

Indicates an error condition has arisen with respect to Syncdrive's handling of the com_block structure. The application must set this field to zero before starting the transmission, and should not attempt to modify its contents after the transmission has started.

If the RX_OVERWRITTEN bit (0x01) is set, this com_block was overwritten when a ring mode receive buffer queue overran and the overrun was permitted. See page 51 for details.

3.6.6 misc_options

If bit 0 of this field is set to 1, the block mode byte streaming feature is enabled. When the end of the *comm_buffer* is reached,

buffer_pointer is reset to 0 and the data in the buffer is overwritten. Syncdrive does not try to detect the *block_term* character when byte streaming. Reception of the frame can be stopped by calling sync_command(4). The *message_type* field must be set for block mode for this feature to be enabled. All other bits in this field are reserved for future use and should be set to 0.

3.6.7 message_type

The type of frame to be received. If *message_type* is 0, the data block will be received using byte synchronous message mode. If *message_type* is any non-zero value, the data block will be received using byte synchronous block mode.

3.6.8 num_pads

The number of pad characters to be received before determining that the communication line has entered the idle state. In block mode, this number of block_term characters must be seen consecutively for the message to terminate.

3.6.9 pad_count

Used by Syncdrive to record the number of pad characters already received. The application may inspect *pad_count* at any time but MUST NOT modify its contents after the reception has started.

3.6.10 block_term

Used by Syncdrive as the block mode termination (pad) character. In block mode receive, the message reception will be terminated when this character is received *num_pads* times consecutively.

3.6.11 reserved_char_x

Reserved for future use. All reserved_char fields must be set to 0.

3.6.12 comm_buffer[ ]

Unsigned character array containing the data to be received. The size of *comm_buffer* is user defined with the restriction that the entire com_block structure must reside in a single 64 Kbyte segment. The size of *comm_buffer* is configured at compile time by the buffer_size define statement (default 1000 bytes) found in the SYNCDRIV.H include file.

3.7 The BufferQueue Structure

The BufferQueue structure is used to allow a Syncdrive application to transfer multiple data frames with one Syncdrive function call. A buffer queue is comprised of a number of com_block structures and the control information contained in a BufferQueue structure, shown in Figure 10.

The application must allocate the com_block structures to be used by the queue. BufferQueue structures are created by Syncdrive when the application calls the sync_register_queue function. BufferQueue structures are deallocated by calling the sync_free_queue function. A BufferQueue structure should be considered read-only by the application.

> IMPORTANT!
>
> The application may inspect the BufferQueue structure at any time but MUST not modify its contents after a transmit or receive operation has been started.

The BufferQueue structure and the queue-related Syncdrive function prototypes are defined in the SYNCDRIV.H include file.

```
struct BufferQueue
      {
      unsigned long NumberOfPointers;     // Number of com_block pointers in this
                                          //    structure.
      unsigned long CurrentBuffer;        // Buffer number (starting with 0) that
                                          //    Syncdrive is currently processing.
      unsigned long NotifyThreshold;      // Reserved.  DO NOT MODIFY!
      unsigned long QueueStatus;          // Overall status of the buffer queue.
      unsigned long QueueMode;            // Oneshot or ring.

      union com_block _far * (_far *ComBlockPtrs);          // Reserved.  DO NOT
MODIFY!
      union com_block _far * (_far *res1);          // Reserved.  DO NOT MODIFY!

      #ifdef __OS2__
      void   _far * (_far *LockHandlePtrs);                 // Reserved.  DO NOT
MODIFY
```

Figure 10 - BufferQueue structure

3.7.1 QueueHNDL

Present only in the Windows version of Syncdrive. It is used internally by Syncdrive and is a reserved field that must not be used by the application.

3.7.2 NumberOfPointers

The number of com_block structures that the buffer queue is managing.

3.7.3 CurrentBuffer

The com_block structure currently being processed by Syncdrive. The first structure is designated as buffer 0. The last structure is buffer (*NumberOfPointers* - 1). The application should monitor the value of *CurrentBuffer* in order to determine when to take action as frames are processed by Syncdrive.

3.7.4 NotifyThreshold

Reserved for future use.

3.7.5 QueueStatus

Bitmapped status indicators, defined in the SYNCDRIV.H include file as follows:

QUEUE_DONE	Queue is stopped. If no error bits are set, all buffers have been processed.
QUEUE_RUNNING	Queue is active. *CurrentBuffer* indicates the com_block currently being processed.
QUEUE_HALTED	Queue was stopped because the application called sync_command(4) or sync_abort_transmit. The most recent frame is likely to be incomplete.
QUEUE_ADVANCED	For Syncdrive internal use only.
QUEUE_OVERRUN	Queue's supply of available com_blocks was exhausted.
QUEUE_ERROR	Generic error indicator.
QUEUE_NULL_BUFFER	Com_block buffer_length of 0 found.

ALL OTHER BITS ARE RESERVED FOR FUTURE USE!

3.7.6 QueueMode

Bitmapped options, described in the SYNCDRIV.H include file, that indicate the operating modes of the queue.

QUEUE_RING	Queue is running in a continuous ring mode until stopped by the application or until a QUEUE_OVERRUN status occurs. *CurrentBuffer* will wrap to buffer 0 when the end of the queue is reached.
QUEUE_ONESHOT	Queue is running in a oneshot mode. All com_block buffers will be processed one time.
QUEUE_RECEIVE	Queue is being used for receive.
QUEUE_TRANSMIT	Queue is being used for transmit.
QUEUE_ALLOW_OVERRUN	Allow com_block reuse or overwrite on queue overrun.

ALL OTHER BITS ARE RESERVED FOR FUTURE USE!

3.7.7 ComBlockPtrs

Pointer to an array of far pointers to com_block structures. This pointer and the array it references should never be changed by the application.

3.7.8 res1

Reserved pointer. The application should not reference or change this pointer.

3.7.9 LockHandlePtrs

Present only in the OS/2 version of Syncdrive. It is used internally by Syncdrive and is a reserved field that must not be used by the application.

4 Operational Overview

Listed below is an outline of the basic steps that an application must take to transmit or receive data frames using Syncdrive.

1. Allocate data structures.
2. Configure the channel.
3. Register buffer queues (optional).
4. Transmit or receive data.
5. Free buffer queues (if used).
6. Release the channel.
7. Deallocate data structures.

4.1 Allocate data structures

4.1.1 Channel Configuration Structure

Each channel to be used requires its own channel configuration structure. This structure is discussed on page 12.

The channel configuration structure(s) may be defined as static data or may be allocated dynamically using standard memory allocation procedures. The structures may be initialized using standard 'C' language conventions at declaration time if a fixed configuration is desired, or during program execution. The application must ensure that the entire structure is initialized.

4.1.2 Com_block Structures

Each com_block structure provides one transmit or receive buffer which can contain one frame worth of data. The com_block structure is discussed beginning on page 23. The application can allocate as many com_block structures as it needs, limited by the amount of available system memory.

There are different protocol-specific versions of the com_block structure that can be used by the application. The various versions are defined as a UNION. The application should use the appropriate protocol-specific version of the com_block structure.

The com_block structure may be statically allocated by declaring a union of type com_block, or may be dynamically allocated by having the malloc function return a pointer to a union of type com_block. These two methods will work if Syncdrive operation is going to occur completely under interrupt control.

If DMA is used for transmitting or receiving frames (non-zero values in tx_dma_channel or rx_dma_channel in the channel configuration structure), the com_block structures MUST be allocated using the sync_alloc_dma_buffer command.

4.2 Configure the channel

A communications channel must be configured before it can be used to transmit or receive data. This is Syncdrive's "logical open" step. To configure a channel, the application must allocate and initialize a channel configuration structure. The channel is uniquely identified by the *board_number* and *channel_number* variables in the channel configuration structure.

The channel is configured by passing a pointer to the channel configuration structure to the config_MPAxxxx function. The actual name of this function is dependent on the particular MPA-series adapter being used. If the config_MPAxxxx function returns the SYNC_SUCCESS code, the channel is ready for use.

Note, once a channel is configured, the channel configuration structure must not be modified until the sync_release function has been executed. Failure to follow this rule will cause improper channel operation.

4.3 Register buffer queues (optional)

Use of buffer queues is optional. If the application plans to use buffer queues, it should register them at this point. Each channel can have one registered buffer queue for transmit and one registered buffer queue for receive. Attempts to register multiple queues will be rejected.

4.4 Transmit and / or receive data

After configuration, the channel is ready for data traffic. Syncdrive can operate in two different ways. The "classic" Syncdrive operation is one frame at a time, using the sync_transmit and sync_receive functions. It is also possible to process multiple frames with one operation by using buffer queues in conjunction with the sync_transmit_queue and sync_receive_queue functions.

4.4.1 One frame at a time

The sync_receive and sync_transmit functions initiate the reception and transmission of single data frames, using a single com_block structure.

To transmit a block of data, the application copies the data to be transmitted into the *comm_buffer* array in the com_block structure and sets the *buffer_status* variable to zero. The application then calls the sync_transmit function. The sync_transmit function immediately returns to the caller while Syncdrive transmits the frame in the background.

The application must monitor the *buffer_status* variable in order to determine when the frame has been completely transmitted. When *buffer_status* indicates TxDone, then the information that was stored in the *comm_buffer* array has been transmitted. The application should also check *buffer_status* for any errors reported.

To receive data, the application ensures that the *buffer_status* variable in the com_block structure is set to zero and then calls the sync_receive function. The sync_receive function immediately returns to the caller while Syncdrive receives the frame in the background.

The application must monitor the *buffer_status* variable in order to determine when the frame has been completely received. When *buffer_status* indicates RxDone, the received data can be transferred out of the *comm_buffer* array and passed on to the application program. The application should also check *buffer_status* for any errors reported.

> IMPORTANT!
>
> The contents of the com_block structure must not be modified while a transmission or reception is in progress!

The parameters that are part of the com_block structure may be modified before the sync_transmit or sync_receive functions are called and after *buffer_status* has indicated that the frame is done. This capability could be useful for protocols such as BISYNC in that Syncdrive allows for dynamic alteration of some protocol parameters. For example, the application could change between message and block mode immediately preceding the starting of reception or transmission of a buffer. Also, the changing of special characters in the BISYNC receive mask may be done just prior to the start of data reception.

Under OS/2, the sync_transmit and sync_receive functions can be made to block until the frame is completed. This is useful in multithreaded programs because it allows an application to eliminate the polling of the *buffer_status* variable.

4.4.2 Using buffer queues

The sync_receive_queue and sync_transmit_queue functions initiate the reception and transmission of multiple data frames, using an array of com_block buffers. Buffer queues must be registered before these functions can be successfully called.

Please refer to the chapter on Buffer Queues, starting on page 47, for full details on how to use this feature.

4.5 Free buffer queues (if used)

If any buffer queues were registered using the sync_register_queue function, the sync_free_queue function must be called for each queue. Freeing the buffer queue does not deallocate any of the member com_block structures.

4.6 Release the channel

The sync_release function is used to reset the hardware, unhook interrupts, and disable all Syncdrive operations. This is Syncdrive's "logical close" step. The application must not attempt to use the channel again after releasing it.

It is best if the application ensures that all communications activity is stopped before calling the sync_release function. This can be accomplished by calling the sync_abort_message and sync_command(4) functions as appropriate.

4.7 Deallocate data structures

Before terminating, the application should free all channel configuration and com_block structures that it has allocated.

If any com_block structures were allocated using the sync_alloc_dma_buffer function, the sync_free_dma_buffers function must be called to free them.

5 Buffer Queues

Syncdrive's normal mode of operation processes one com_block structure at a time. For applications where frames appear on the communications link with short inter-frame intervals, this can be a problem, especially on the receive side. The application may receive a frame successfully, but may not be able to issue another call to sync_receive before the next frame starts to come in. Less often, on the transmit side, the same difficulty can cause unacceptable idle periods between frames.

Buffer queues allow the application to setup multiple com_block structures ahead of time and have Syncdrive process them in a batch. When a buffer queue is used, after a frame is completely processed, Syncdrive will immediately start to process the next com_block structure.

A buffer queue is essentially a list of com_block structures. Each com_block structure in the buffer queue is allocated and accessed by the application in the same fashion as a "stand alone" com_block structure would be.

The operation of the buffer queue is configured and monitored through a BufferQueue structure (see page 37). The application does not directly allocate the BufferQueue structure.

5.1 Creating buffer queues

The application must determine how many frame buffers will be in the buffer queue, and then allocate one com_block structure for each buffer. The pointers to these com_block structures are placed in an array. The com_block pointer array and some control information are passed as parameters to the sync_register_queue function. The sync_register_queue function creates and initializes a BufferQueue structure. A pointer to the BufferQueue structure is returned to the application.

5.2 Disposing of buffer queues

After the application is finished with the buffer queue, the BufferQueue structure is deregistered and deallocated by calling the sync_free_queue function. The application should ensure that communications activity is stopped before freeing the buffer queue. This can be done by calling the sync_abort_transmit and sync_command(4) functions as necessary before calling sync_free_queue.

5.3 Restrictions imposed by buffer queues

When the sync_register_queue function is used to create and register a buffer queue, the operating modes of the buffer queue are fixed. If an operating mode needs to be changed, the buffer queue must be freed and a new one registered.

Each channel can have one buffer queue registered for transmit and one buffer queue registered for receive. The transmit and receive buffer queues are completely independent of each other, so it is possible to use a buffer queue in one direction but not use one in the other direction.

It is not possible to mix buffer queue and single-frame operation on the fly. If a buffer queue is registered for receive operation, calls to the sync_receive function will be rejected. If a buffer queue is not registered for receive operation, calls to the sync_receive_queue function will be rejected. The same rule applies to transmit operations.

5.4 Transmitting or receiving data with a buffer queue

5.4.1 Setup the buffer queue

The application must allocate a sufficient number of com_block structures. The *buffer_status* variable in each com_block structure must be set by the application to 0. For transmit, the data for each frame must be placed into the *comm_buffer* arrays in the various com_block structures. The buffer queue should then be registered by calling the sync_register_queue function.

5.4.2 Tell Syncdrive to start processing frames

The sync_transmit_queue function initiates the transmission of the data frames contained in the *comm_buffer* arrays of the com_block structures in the registered transmit buffer queue. Syncdrive continuously transmits frames until the queue is exhausted or stopped. The sync_transmit_queue function immediately returns to the caller while Syncdrive processes the queue in the background.

The sync_receive_queue function initiates the reception of data frames. Frame data will be placed in the *comm_buffer* arrays of the com_block structures in the registered receive buffer queue. Syncdrive continuously receives frames until the queue is filled or stopped. The sync_receive_queue function immediately returns to the caller while Syncdrive processes the queue in the background.

IMPORTANT!

The contents of the BufferQueue structure must not be modified while a transmission or reception is in progress!

5.4.3 Monitor Syncdrive's progress

The application should monitor the *QueueStatus* variable in the BufferQueue structure. *QueueStatus* will indicate whether the buffer queue is still running and whether any error conditions have been flagged by Syncdrive.

The application can also check the *CurrentBuffer* variable in the BufferQueue structure. As *CurrentBuffer* increments, it indicates that previous frames have been completely transmitted or received. If the buffer queue is running in ring mode, *CurrentBuffer* will wrap to zero when a new cycle through the queue begins.

The variables within a com_block structure are updated during queue operation as a frame is processed. After the frame is completed, the application should check for errors in the com_block structure's *buffer_status* variable just as it would in "one frame at a time" mode.

5.4.4 Oneshot mode

In oneshot mode, buffer queue operation will stop after the last com_block structure is processed. In this mode, the application can check for a QUEUE_DONE indication in the *QueueStatus* variable in the BufferQueue structure to determine when the entire buffer queue has been processed.

5.4.5 Ring mode

In order to maintain a running buffer queue in ring mode, the application must refresh the com_block structures as they are processed by Syncdrive. The application must process frame data as necessary and set the *buffer_status* variable to 0 before Syncdrive will attempt to reuse that com_block structure.

Syncdrive will not process a com_block structure whose *buffer_status* variable contains a non-zero value. Instead, ring mode operation will terminate with a QUEUE_OVERRUN indication in the *BufferStatus* variable of the BufferQueue structure.

5.4.6 Allowing queue overruns

As an enhancement to ring mode, Syncdrive can optionally tolerate buffer queue overruns. If QUEUE_ALLOW_OVERRUN is set in the *QueueMode* variable, Syncdrive will continue processing a buffer queue when it encounters com_block structures with non-zero *buffer_status* variables. The QUEUE_OVERRUN indication in the *BufferStatus* variable of the BufferQueue structure will be set, but the QUEUE_RUNNING indicator will also remain set.

For transmit buffer queues, the data in the com_block structure will be retransmitted, with the *buffer_status* variable being cleared just before the buffer is reused. Syncdrive will set the TX_RETRANSMITTED bit in the *buffer_error* variable. This status will remain set until the application clears it. No count is kept of the number of times the com_block is reused. For receive buffer queues, the data in the com_block structure will be overwritten, with the *buffer_status* variable being cleared just before the buffer is reused. Syncdrive will set the RX_OVERWRITTEN bit in the *buffer_error* variable. This status will remain set until the application clears it. No count is kept of the number of times the com_block is reused.

5.5 Stopping a buffer queue

The sync_abort_message and sync_command(4) functions will stop the transmit or receive operation, respectively, on the current com_block structure. The action taken on the active buffer is precisely the same as in the case of a single-frame transmission or reception being stopped.

In addition, the buffer queue will be stopped, and the QUEUE_HALTED indicator will be set in the *BufferStatus* variable of the BufferQueue structure. The application can then use the *CurrentBuffer* variable in the BufferQueue structure and the *buffer_status* variable of the individual com_block structures to determine the progress that had been made up to the time the buffer queue was stopped.

5.6 Reusing a buffer queue

To reuse a halted or completed buffer queue, the application must simply set the *buffer_status* variable of each com_block structure to 0, load new data into the *comm_buffer* arrays if transmitting, and call the sync_transmit_queue or sync_receive_queue function again.

It is not necessary to free and reregister the buffer queue unless the application wishes to change the operating modes of the queue.

5.7 Buffer queue performance

Buffer queues perform best when DMA is used. Keeping the overall interrupt load on the system as low as possible is encouraged. Because of the heavier interrupt load of byte-synchronous modes, buffer queues may not work as well as they do in bit-synchronous modes.

It can also be helpful to avoid using some of the more cumbersome runtime library functions such as printf while a buffer queue is running. Under DOS, runtime library functions seem to disable interrupts for long periods of time. This behavior can disrupt queue operation.

Transmit buffer queues run very fast. Unless a large number of buffers is used, the application may have trouble keeping up with the speed of the queue in ring mode. The application should be prepared to deal with the QUEUE_OVERRUN status accordingly.

6 OS/2 Operation

The OS/2 version of Syncdrive is very similar to the DOS version in its feature set and general procedures for use. There are a number of issues, however, of which the developer should be aware.

6.1 Application compilation

Applications must use the SYNCDRIVE_API calling convention, defined in the MPA-X00.H include file, in order to access the DLL. Structures must be packed for byte alignment. The SYNCOS2.H include file must be included before any other Syncdrive include file. It contains definitions needed by the other Syncdrive include files.

The "__OS2__" macro must be defined when compiling applications. Most OS/2 compilers will automatically define this macro. If it is not defined, the SYNCOS2.H include file will define it.

6.2 Buffer memory allocation

If and only if DMA is used to transmit or receive frames, com_block structures MUST be allocated with the sync_alloc_dma_buffer function.

6.3 Application termination

Syncdrive for OS/2 locks memory ranges during its operation. If the application terminates without this memory being released, it may block further use of the Syncdrive facilities, and may result in an uncloseable session, necessitating a shutdown and reboot. Therefore, **an application must ensure that the sync_release function is called before exiting for any reason.** The application should call sync_release before freeing com_block structures.

If a transmission or reception is in progress at termination, it is a good idea to call sync_abort_message or sync_command(4) before calling sync_release.

The sync_release function ensures that all locked memory is unlocked, except for DMA buffers. **If DMA buffers were allocated, the sync_free_dma_buffers function must be called.** Because applications can terminate unexpectedly due to bugs or operator intervention, sync_free_dma_buffers and sync_release should be called in exit handler and signal handler functions.

6.4 Multithreaded applications

The *read_timeout* element of the channel configuration structure sets the number of milliseconds that the device driver will wait for a frame to be completely received. A thread calling sync_receive will be blocked in the device driver if this timeout value is non-zero. The thread will be run again when the end-of-frame condition occurs or when the *read_timeout* interval expires, whichever happens first.

If the *read_timeout* value is zero, calls to sync_receive will return immediately. In this mode, which allows for easy porting of DOS Syncdrive applications to OS/2, the application needs to poll the *buffer_status* variable in the com_block structure to determine when reception of a frame has been completed. Conversely, if the *read_timeout* is -1 (0xFFFFFFFF), the thread will wait forever for a frame to be received.

The *write_timeout* element of the channel configuration structure is the timeout for the sync_transmit function, providing a time limit for the completion of a frame transmission. It is independent of the *read_timeout*. While a thread is blocked on transmit or receive in the device driver, that portion of the device driver is considered busy.

Frame timeouts are indicated in the *buffer_status* variable in the com_block structure. The device driver will abort the frame as if the application had called sync_abort_message or sync_command(4). The data is lost, and the transmitter or receiver is reset. The application should always check the *buffer_status* variable in the com_block structure when a call to sync_transmit or sync_receive returns to see if the reason for completion was a timeout.

The sync_transmit and sync_receive functions themselves also have a *timeout* parameter. If the device driver is busy with a pending operation, the function will block on a mutex semaphore if the *timeout* parameter is non-zero. If this timeout occurs before the device driver becomes non-busy, or if the *timeout* parameter is zero, the function itself will return a timeout error status. If the *timeout* value is -1, the function will wait forever for access to the device driver.

If the *read_timeout* and *write_timeout* values in the channel configuration structure are zero, the device driver will never appear busy to the DLL. This means that threads will not block in the DLL either, and the application must take responsibility for serializing access to the hardware or risk loss of data by having threads overwriting each other's data.

6.5 Buffer queues

At this time, Syncdrive for OS/2 does not support blocking threads on queue operations. OS/2 applications must monitor the indicators in the BufferQueue structure to track the queue's progress. Accordingly, the *read_timeout* and *write_timeout* values in the channel configuration array are ignored for queue operations.

6.6 Pointers

All pointers in Syncdrive for OS/2 are 32-bit flat pointers. 16-bit access to the DLL is not supported. The SYNCOS2.H include file nullifies the "near" and "far" keywords for easy porting of DOS Syncdrive applications.

6.7 Multiple processes

While Syncdrive for OS/2 supports multithreaded applications, it cannot at this time support multiple processes accessing the DLL.

7 Windows 3.1/95/98 Operation

The Windows 3.1 and Windows 95/98 versions of Syncdrive are very similar to the DOS version in their feature set and general procedures for use. There are a number of issues, however, of which the developer should be aware.

7.1 Windows NT (a.k.a. Windows 2000)

This version of Syncdrive supports Microsoft Windows 3.1, Windows 95, and Windows 98. It does not support Microsoft Windows NT.

7.2 Application compilation for Windows 3.1

Applications for Windows 3.1 must be compiled for large model, and must include the SNC31DLL.H file. Applications must use the SYNCDRIVE_API calling convention, defined in the MPA-X00.H include file, in order to access the Windows DLL.

Structures must be packed for byte alignment. The SNC31DLL.H include file must be included before any other Syncdrive include file. It contains definitions needed by the other Syncdrive include files.

The "_WINDOWS" macro must be defined when compiling applications. Most Windows compilers will automatically define this macro. If it is not defined, the SNC31DLL.H include file will define it.

7.3 Application compilation for Windows 95/98

Windows 95/98 applications use flat model and must include the SNC95DLL.H file. Applications must use the SYNCDRIVE_API calling convention, defined in the MPA-X00.H include file, in order to access either the Windows DLL.

Structures must be packed for byte alignment. The SNC95DLL.H include file must be included before any other Syncdrive include file. It contains definitions needed by the other Syncdrive include files.

The "_WINDOWS" and "__WIN95__" (two underscores on each side) macros must be defined when compiling applications. If they are not defined, the SNC95DLL.H include file will define them.

7.4 Buffer memory allocation

If and only if DMA is used to transmit or receive frames, all com_block structures MUST be allocated with the sync_alloc_dma_buffer function.

Starting with Syncdrive release 4.00, there is no longer a need to allocate non-DMA com_block structures with locked memory. A simple malloc() call should suffice. Syncdrive will automatically lock memory as needed.

7.5 Application termination

Applications should ensure that any locked memory is freed before terminating. DMA buffers must be freed with the sync_free_dma_buffers function. The application should call sync_release before freeing com_block structures. It is a good idea to call sync_abort_message or sync_command(4) before calling sync_release.

Because applications can terminate unexpectedly due to bugs or operator intervention, termination procedures should be called in exit handler functions.

7.6 Multithreaded applications

Windows 3.1 does not support multithreaded applications. The *read_timeout* and *write_timeout* values in the channel configuration array should be set to zero. Windows applications must also specify a *timeout* parameter of zero on sync_transmit and sync_receive calls.

Windows 95/98 does support multithreaded applications, but unfortunately Syncdrive does not yet accommodate them. 32-bit Windows 95/98 applications should treat the various timeout values in the same manner proscribed for Windows 3.1 applications.

Thread support similar to that in the OS/2 version of Syncdrive is envisioned for a future release of the Windows 95/98 Syncdrive code.

7.7 Completion of frames

Applications must wait for frames to complete transmission or reception by monitoring the *buffer_status* field in the com_block structure. This operation is identical to the DOS version of Syncdrive and the polled mode of the OS/2 version. There are currently no mechanisms in place for callbacks or messages to an application at end-of-frame.

7.8 Buffer queues

Applications must monitor the various indicators in the BufferQueue structure to track the queue's progress.

7.9 Pointers

All pointers in Syncdrive for Windows 3.1 are 16:16 far pointers.

All pointers in Syncdrive for Windows 95/98 are 32-bit flat pointers. The SNC95DLL.H include file nullifies the "far" keyword for easy porting of DOS Syncdrive applications.

7.10 Windows 3.1 applications on Windows 95/98

In order to provide for backward compatibility, Syncdrive supports running 16-bit Windows 3.1 Syncdrive applications on Windows 95/98. In order to do this, the Windows 3.1 version of Syncdrive (release 4.00 or later) must be installed on Windows 95/98. Follow the directions for installation of Syncdrive for Windows 3.1 given on page 2.

The Windows 3.1 version of Syncdrive will not execute 32-bit Syncdrive applications. The Windows 3.1 version must be removed before using the native 32-bit version of Syncdrive for Windows 95/98. The two versions cannot coexist.

8 Syncdrive Application Programming Interface

There are a number functions that make up the application programs entry points into Syncdrive. These Syncdrive API functions are listed below. All Syncdrive API functions are prototyped in the MPA-X00.H Syncdrive include file.

- **config_MPA100** - configure an MPA-100 channel
- **config_MPA200** - configure an MPA-200/300 channel
- **config_MPA102** - configure an MPA-102 channel
- **config_MPAP100** - configure an MPAP-100 channel
- **sync_release** - release channel configuration
- **sync_transmit** - transmit a frame
- **sync_receive** - receive a frame
- **sync_register_queue** - create a buffer queue structure
- **sync_free_queue** - destroy a buffer queue structure
- **sync_transmit_queue** - transmit a buffer queue of frames
- **sync_receive_queue** - receive a buffer queue of frames
- **sync_abort_message** - abort the current buffer transmission
- **sync_command** - execute various Syncdrive commands
- **sync_alloc_dma_buffer** - allocate a DMA buffer. (Bit synchronous modes only.)
- **sync_free_dma_buffers** - deallocate all DMA buffers

The following functions are obsolete and are no longer documented, but are supported in the DLL for backward compatibility:

- **sync_lmem_alloc** (Windows 3.1 only)
- **sync_lmem_free** (Windows 3.1 only)

  This pair of functions can be replaced with calls to malloc() and free(), or equivalent operating system specific functions.

8.1 Calling Convention

All Syncdrive functions use a C-language interface with the SYNCDRIVE_API call linkage macro. This macro is defined in the MPA-X00.H include file for the various operating systems. Using it can help make Syncdrive applications more portable from one operating system to another.

8.2 Return Values

All Syncdrive functions return an integer value. A successful completion is indicated by a return code of SYNC_SUCCESS (0).

If Syncdrive cannot successfully complete a function, it will return a non-zero error code. The error codes are defined in the SYNCDRIV.H include file and are also described on page 103.

Error codes prefixed with "FATAL" indicate difficulties that Syncdrive has encountered which are usually beyond the application's control. Actions encountering "FATAL" errors should usually be retried because they may occur as a result of a temporary condition. "FATAL" errors consistently encountered should be reported to Quatech's Technical Support department.

8.3 Examples

Use of the Syncdrive API functions is demonstrated in the numerous example programs supplied on the Syncdrive diskette set. These example programs are described beginning on page 87.

8.4 **config_MPA100** - Open a communications channel
config_MPA200
config_MPA102
config_MPAP100

```
short SYNCDRIVE_API config_MPA100(struct channel_cfg _far
*pMPA_Cfg); (other functions are similar)
```

The config_MPAxxx function is the Syncdrive channel configuration function. The application must execute the function corresponding to the particular MPA-series adapter being used. Most other Syncdrive functions require the channel to already be configured.

The channel configuration structure pointed to by pMPA_Cfg defines the hardware configuration and operating mode of the installed communication adapter. Syncdrive uses this information to initialize the hardware, hook interrupts, and prepare itself for data transfers.

NOTE: Use config_MPAP100() for the MPAP-200/300 and MPAC-100 adapters.

8.4.1 pMPA_Cfg

Pointer to the channel configuration structure defining the configuration and operating mode of the communication channel. The application must allocate and initialize this structure prior to calling this function.

IMPORTANT!

Once a channel is configured, the application must not make any changes to the channel configuration structure. If changes are required, the channel must be released and reconfigured.

8.5 **sync_release** - Close a communications channel

```
short SYNCDRIVE_API sync_release(struct channel_cfg _far
*pMPA_Cfg);
```

The sync_release function is the reciprocal of the config_MPAxxx function. The hardware is reset, interrupts are unhooked, and all Syncdrive operations are disabled. The sync_release function must be called to release a channel's configuration before a new channel configuration can be set.

8.5.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

> IMPORTANT!
>
> The sync_release function must be executed before the application exits. Before calling sync_release, the application should call the sync_command(4) and / or sync_abort_message functions to stop pending data transfers.

8.6 **sync_transmit** - Transmit a single frame of data

(DOS)

```
short SYNCDRIVE_API sync_transmit(  struct channel_cfg _far
*pMPA_Cfg,                          union com_block _far
*pTxBuffer);
```

(OS/2 and Windows)

```
short SYNCDRIVE_API sync_transmit(  struct channel_cfg *pMPA_Cfg,
                                    union com_block *pTxBuffer,
                                    unsigned long timeout);
```

This function starts transmission the frame of data in the *comm_buffer* array element of the com_block structure. The channel must be configured prior to calling sync_transmit.

The sync_transmit function returns immediately to the caller (unless using the *write_timeout* feature under OS/2). Syncdrive completes the transmission of the frame in the background using interrupts and/or DMA. The application may check the status of the transmission using the *buffer_status* variable in the com_block structure.

8.6.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.6.2 pTxBuffer

Pointer to a com_block structure defining the message dependent communication parameters and containing the data block to be transmitted.

8.6.3 timeout (OS/2 and Windows)

Unsigned long integer specifying the number of milliseconds to wait when the device driver is busy transmitting a frame. If zero, sync_transmit will return immediately if the driver is busy. If -1, sync_transmit will wait forever for the driver to become non-busy. This value should be zero for Windows, but is currently ignored.

8.7 sync_receive - Receive a single frame of data

(DOS)

```
short SYNCDRIVE_API  sync_receive(  struct channel_cfg _far
*pMPA_Cfg,                          union com_block _far
*pRxBuffer);
```

(OS/2 and Windows)

```
short SYNCDRIVE_API  sync_receive(  struct channel_cfg *pMPA_Cfg,
                                    union com_block *pRxBuffer,
                                    unsigned long timeout);
```

This function starts the reception of one frame of data and places the data in the *comm_buffer* array element of the com_block structure. The channel must be configured prior to calling sync_receive.

The sync_receive function returns immediately to the caller (unless using the *read_timeout* feature under OS/2). Syncdrive completes the reception of the frame in the background using interrupts and/or DMA. The application may check the status of the reception using the *buffer_status* variable provided in the com_block structure.

8.7.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.7.2 pRxBuffer

Pointer to a com_block structure defining the message dependent communication parameters and containing the data block where received data is to be stored.

8.7.3 timeout (OS/2 and Windows)

Unsigned long integer specifying the number of milliseconds to wait when the device driver is busy receiving a frame. If zero, sync_receive will return immediately if the driver is busy. If -1, sync_receive will wait forever for the driver to become non-busy. This value should be zero for Windows, but is currently ignored.

8.8 **sync_abort_message** - Abort current transmission

```
short SYNCDRIVE_API sync_abort_message(struct channel_cfg _far
*pMPA_Cfg);
```

This function causes an abort to be signaled on the communication line. Transmit interrupts and DMA are disabled. If a message is being sent when sync_abort_message is called, the com_block structure is preserved so the application can determine what the progress had been up to that time.

The sync_abort_message function will stop a running transmit buffer queue. The current com_block structure will be treated normally and the *BufferStatus* variable in the BufferQueue structure will have the QUEUE_HALTED indicator set.

8.8.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.9 **sync_alloc_dma_buffer** - Allocate a DMA buffer

```
short SYNCDRIVE_API sync_alloc_dma_buffer(struct channel_cfg far
*pMPA_Cfg,                                unsigned short size,
                                          union com_block far
*(*pBuffer));
```

The sync_alloc_dma_buffer routine is used to allocate a com_block structure when DMA is to be used to transmit or receive data. Use of this function guarantees that the memory buffer will not cross a DMA page boundary. Crossing a DMA page boundary causes an error during a call to sync_transmit or sync_receive preventing the initialization of frame transmission or reception.

The buffers allocated by this routine must be deallocated by calling the sync_free_dma_buffers function!

> IMPORTANT!
>
> Use this function ONLY to allocate DMA com_block buffers!
> Do not use it for any other memory allocations!

8.9.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.9.2 size

An unsigned integer value defining the size of the buffer to be allocated. The actual size of the buffer will be slightly less than this because of buffer overhead for *buffer_status* and other elements of the com_block structure.

8.9.3 pBuffer

Pointer to a com_block structure. Note that **the actual parameter passed to the function is the address of this pointer**. When sync_alloc_dma_buffer returns, pBuffer will contain a pointer to the newly allocated com_block structure.

8.10 **sync_free_dma_buffers** - Free all DMA buffers

```
short SYNCDRIVE_API sync_free_dma_buffers(void);
```

This function deallocates all DMA buffers previously allocated with the sync_alloc_dma_buffer function. It cannot free individual DMA buffers.

This function has no parameters.

> IMPORTANT!
>
> The application must free all DMA buffers before exiting!

8.11 sync_command - Issue a miscellaneous command

```
short SYNCDRIVE_API sync_command(struct channel_cfg _far
*pMPA_Cfg,                          short command,
                                    ...);
```

The sync_command function provides a number of services that allow the application to define parameters independent of the channel configuration and to control functions of the communication hardware that are not otherwise supported by Syncdrive. When calling the sync_command function, the service provided is determined by the value of the *command* parameter. Descriptions of each sync_command service appear on the next several pages.

The sync_command function takes a variable number of parameters, dependent on the particular service being requested.

8.11.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.11.2 command

An integer variable specifying the service to be executed. The following table lists the valid *command* values.

Command	Description
0	get communication controller status
1	set communication controller status
2	set transmitter baud rate
3	set receiver baud rate
4	reset receiver
5	modify byte sync receive mask

8.11.3 other parameters

The number and types of parameters are dependent on the service selected. See the following pages for the specific services.

8.12 sync_command(0) - Get Communications Status

```
result = sync_command(pMPA_Cfg, 0, pChannelStatus);
```

The get communications controller status command allows the application program to examine the current state of the modem control lines, the output control lines, and the communication mode (hunt or receive data mode).

8.12.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.12.2 pChannelStatus

Pointer to an unsigned integer variable used to return the status of the communication controller to the application program. The format of scc_status is shown below:

D15	D14	D13	D12	D11	D10	D9	D8
DTR	RTS	CTS	DCD	DSR	0	0	0

D7	D6	D5	D4	D3	D2	D1	D0
OE[1]	HUNT	0	0	0	0	0	0

- **DTR** - indicates the current state of the DTR output.
- **RTS** - indicates the current state of the RTS output.
- **CTS** - indicates the current state of the CTS input.
- **DCD** - indicates the current state of the DCD input.
- **DSR** - indicates the current state of the DSR input.
- **OE[1]** - indicates the current state of the communication driver outputs. When set to 1, the transmit drivers are enabled.

[1] OE is valid only with communication hardware featuring tri-state output drivers.

- **HUNT** - when set to 1, the communication controller is in the hunt mode. When set to 0, the controller is in the receive data mode.

8.13 sync_command(1) - Set Communications Controller State

```
result = sync_command(pMPA_Cfg, 1, ChannelStatus);
```

The set communications controller state allows the application program to define the state of the modem control and output driver control lines.

8.13.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.13.2 ChannelStatus

An unsigned integer variable defining the state of the modem control lines of the channel and SCC status. The format of scc_config is shown below:

D15	D14	D13	D12	D11	D10	D9	D8
DTR	RTS	0	0	0	0	0	0

D7	D6	D5	D4	D3	D2	D1	D0
OE[2]	HUNT	-	-	-	-	-	-

- **DTR** - specifies the state of the DTR output.
- **RTS** - specifies the state of the RTS output.
- **OE[2]** - specifies the state of the communication driver outputs. When set to 1, the output drivers are enabled for transmission.

[2] OE is only valid with communications hardware featuring tri-state driver outputs.

8.14 sync_command(2) - Change Channel "A" Baud Rate

```
result = sync_command(pMPA_Cfg, 2, baud_rate, clock_rate);
```

This command allows the application program to change the baud rate of the SCC's "A" channel after the channel has been configured. In most cases, this will affect the transmit baud rate. Depending on the clock configuration chosen (see the discussion of the *clock_source* variable on page 18), it may also affect the receive baud rate.

> IMPORTANT!
>
> Do not use this function in place of setting the *tx_baud_rate* and *rx_baud_rate* variables in the channel configuration structure! This function is not a replacement for normal configuration methods!

After the baud rate hardware has been reprogrammed, the actual new baud rate is calculated and stored in the *tx_baud_rate* variable in the channel configuration structure. Differences between the actual baud rate and the requested baud rate occur when the baud rate does not divide evenly into the clock source frequency.

8.14.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.14.2 baud_rate

An unsigned long integer variable specifying the desired baud rate.

8.14.3 clock_rate

An unsigned long integer variable specifying the input clock frequency to the communication controller.

8.15 sync_command(3) - Change Channel "B" Baud Rate

```
result = sync_command(pMPA_Cfg, 3, baud_rate, clock_rate);
```

This command allows the application program to change the baud rate of the SCC's "B" channel after the channel has been configured. Depending on the clock configuration chosen (see the discussion of the *clock_source* variable on page 18), it may affect the receive baud rate, the transmit baud rate, or both.

> IMPORTANT!
>
> Do not use this function in place of setting the *tx_baud_rate* and *rx_baud_rate* variables in the channel configuration structure! This function is not a replacement for normal configuration methods!

After the baud rate hardware has been reprogrammed, the actual new baud rate is calculated and stored in the *rx_baud_rate* variable in the channel configuration structure. Differences between the actual baud rate and the requested baud rate occur when the baud rate does not divide evenly into the clock source frequency.

8.15.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.15.2 baud_rate

An unsigned long integer variable specifying the desired baud rate.

8.15.3 clock_rate

An unsigned long integer variable specifying the input clock frequency to the communication controller.

8.16 sync_command(4) - Reset Receiver

```
result = sync_command(pMPA_Cfg, 4);
```

The reset receiver command stops reception of the current message by disabling the receiver, disabling receiver interrupts, and by masking the DMA channel if DMA was being used. If a message is being received when is command is executed, the com_block structure is preserved so the application can determine what the progress had been up to that time.

This command will stop a running receive buffer queue. The current com_block structure will be treated normally and the *BufferStatus* variable in the BufferQueue structure will have the QUEUE_HALTED indicator set.

8.16.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.17 sync_command(5) - Change Byte Sync Receive Mask

```
result = sync_command(pMPA_Cfg, 5, rx_char, rx_type);
```

This sync_command service allows the application program to define the special receive characters for Syncdrive's byte synchronous message mode. The frame's sync, SDI, EDI, and pad characters must all be defined using service 5.

8.17.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.17.2 rx_char

An unsigned character variable specifying the character to be defined in the receive mask.

8.17.3 rx_type

An unsigned character variable specifying the type of character *rx_char* represents:

Value	Meaning
0	Data Character (reset to standard character)
1	Start of Data Indicator (SDI)
2	End of Data Indicator (EDI)
3	Pad Character
4	Sync Character

An *rx_type* of 0 is used to remove the interpretation a special received character and start receiving it as a data character. For example, to change the SDI character from an 0x02 to an 0x08, *rx_type* of 0 would be assigned to the 0x02 character to stop its interpretation as the SDI character. The 0x08 character would then be assigned a type of 1, for SDI. Without removing the SDI interpretation of the 0x02, Syncdrive would interpret both characters as the SDI character.

8.18 sync_register_queue - Create and register a buffer queue

```
short SYNCDRIVE_API sync_register_queue(     struct channel_cfg _far
*pMPA_Cfg,                                   unsigned long NumBuffers,
                                             union com_block _far
*BufferArray[],                         unsigned long threshold,
                                        unsigned long mode,
                                        struct BufferQueue _far *
(*pQueue));
```

This function is used to create a queue of com_block buffers that can be used with the sync_transmit_queue or sync_receive_queue functions. Syncdrive allocates memory for the queue structure and initializes it for use, returning a pointer to the queue structure to the application.

A communications channel may use only one transmit queue and one receive queue at a time. Attempts to register subsequent queues are rejected. A queue is considered registered until it is deregistered with the sync_free_queue function.

The application must allocate all of the com_block structures to be used as part of the buffer queue before registering the queue. If the channel will be run using DMA, the buffers should be allocated using the sync_alloc_dma_buffer function.

8.18.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.18.2 NumBuffers

An unsigned long integer specifying the number of com_block pointers in *BufferArray[]*.

8.18.3 BufferArray[]

An array of pointers to com_block structures. The application must allocate com_block structures and create this array filled with pointers to those structures.

8.18.4 threshold

A reserved unsigned long integer. Set to 0.

8.18.5 mode

Bitmapped values that controls how the buffer queue will operate. The application must select one value from each of the following groupings and OR them together:

QUEUE_TRANSMIT	for use with sync_transmit_queue.
QUEUE_RECEIVE	for use with sync_receive_queue.
QUEUE_ONESHOT	process buffers once, then stop.
QUEUE_RING	process buffers in a continuous ring.

Additional "a la carte" option:

QUEUE_ALLOW_OVERRUN	recycle com_blocks in ring mode.

8.18.6 pQueue

Pointer to a BufferQueue structure. Note that **the actual parameter passed to the function is the address of this pointer**. When sync_register_queue returns, *pQueue* will contain a pointer to the newly registered BufferQueue structure.

8.19 sync_free_queue - Deregister a buffer queue

```
short SYNCDRIVE_API sync_free_queue(struct channel_cfg _far
*pMPA_Cfg,                          struct BufferQueue _far
*pQueue);
```

This function is used to deregister a queue of com_block buffers. Communications activity for the queue should be stopped by calling the sync_abort_transmit or sync_command(4) function before freeing the queue.

The application must free all buffer queues before terminating. It is best to free the queues before closing the communications channel with the sync_release function. As a precaution, however, sync_release will free any registered buffer queues. If sync_free_queue is called after sync_release is called, a QUEUE_NOT_REGISTERED error code is returned.

The application must not reference the freed queue because Syncdrive deallocates the memory for the queue structure. The individual com_block buffers that make up the queue are not affected, and the application may continue to access them after calling this function.

The application is responsible for deallocating all of the com_block structures used as part of the buffer queue. If the channel was run using DMA, the buffers should be deallocated using the sync_free_dma_buffers function.

8.19.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.19.2 pQueue

Pointer to the BufferQueue structure to be freed.

8.20 sync_transmit_queue - Start a transmit buffer queue

(DOS)

```
short SYNCDRIVE_API sync_transmit_queue(  struct channel_cfg _far
*pMPA_Cfg,                                struct BufferQueue _far
*pQueue);
```

(OS/2 and Windows)

```
short SYNCDRIVE_API sync_transmit_queue(  struct channel_cfg
*pMPA_Cfg,                                struct BufferQueue
*pQueue,                                  unsigned long timeout);
```

This function initiates the transmission of the data frame in the first com_block structure in a buffer queue. The buffer queue must have been previously registered using the sync_register_queue function, and must have had the QUEUE_TRANSMIT mode indicated when registered.

After the first buffer has been transmitted, Syncdrive will continue transmitting the queue's other buffers in the background. When the last buffer in the queue has been transmitted, Syncdrive will stop if the QUEUE_ONESHOT mode was chosen when the queue was registered.

If the QUEUE_RING mode was chosen, Syncdrive will attempt to process the queue again, starting with the first buffer. In order for a buffer to be processed again, the *buffer_status* byte of the com_block structure must have been reset to zero by the application. If the *buffer_status* byte is non-zero, a QUEUE_OVERRUN status is generated and the queue operation is stopped.

The application may check the progress of the queue by using the *QueueStatus* and *CurrentBuffer* elements of the BufferQueue structure. The application may also check the status of an individual buffer using the *buffer_status* variable provided in the com_block structure.

8.20.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.20.2 pQueue

Pointer to the BufferQueue structure.

8.20.3 timeout (OS/2 and Windows)

A reserved unsigned long integer. Set to 0.

8.21 sync_receive_queue - Start a receive buffer queue

(DOS)

```
short SYNCDRIVE_API sync_receive_queue(   struct channel_cfg _far
*pMPA_Cfg,                                struct BufferQueue _far
*pQueue);
```

(OS/2 and Windows)

```
short SYNCDRIVE_API sync_receive_queue(   struct channel_cfg
*pMPA_Cfg,                                struct BufferQueue
*pQueue,                                  unsigned long timeout);
```

This function initiates the reception of a data frame and place the received data in the first com_block structure in a buffer queue. The buffer queue must have been previously registered using the sync_register_queue function, and must have had the QUEUE_RECEIVE mode indicated when registered.

After the first buffer has been received, Syncdrive will continue filling other buffers in the queue. When the last buffer in the queue has been filled, Syncdrive will stop if the QUEUE_ONESHOT mode was chosen when the queue was registered.

If the QUEUE_RING mode was chosen, Syncdrive will attempt to process the queue again, starting with the first buffer. In order for a buffer to be processed again, the *buffer_status* byte of the com_block structure must have been reset to zero by the application. If the *buffer_status* byte is non-zero, a QUEUE_OVERRUN status is generated and the queue operation is stopped.

The application may check the progress of the queue by using the *QueueStatus* and *CurrentBuffer* elements of the BufferQueue structure. The application may also check the status of an individual buffer using the *buffer_status* variable provided in the com_block structure.

8.21.1 pMPA_Cfg

Pointer to the channel configuration structure of the communication channel. This must be the same structure that was provided to the config_MPAxxx function.

8.21.2 pQueue

Pointer to the BufferQueue structure.

8.21.3 timeout (OS/2 and Windows)

A reserved unsigned long integer. Set to 0.

9 Building Syncdrive Applications

Each of the platforms that Syncdrive supports has slightly different methods of linking the Syncdrive functions to an application.

- DOS

  Choose the appropriate Syncdrive library from the distribution disks. Link the Syncdrive library with the application's object modules. Only large model is supported.

  The library files can be found in the \DOS directory of disk 1 of the Syncdrive diskette set:

TSYNCDRL.LIB	Borland C large model library
SYNCDRVL.LIB	Microsoft C large model library

- Windows 3.1

  Compile for large memory model.

  Link the application's object modules with the Windows 3.1 Syncdrive API import library file SYNC_31.LIB. This file can be found in the \WIN31 directory of disk 1 of the Syncdrive diskette set.

  Ensure that the Windows 3.1 device driver (VxD) and DLL have been installed properly on the system where the application is to be executed.

- Windows 95/98

  Link the application's object modules with the appropriate Windows 95/98 Syncdrive API import library file. The file can be found in the \WIN95 directory of disk 1 of the Syncdrive diskette set.

SYNCDRIV.LIB	Import library for Borland C++ 5.0
SYNCMSVC.LIB	Import library for Visual C++ 5.0/6.0

  Ensure that the Windows 95 device driver (VxD) and DLL have been installed properly on the system where the application is to be executed.

- OS/2

  Link the application's object modules with the OS/2 Syncdrive API import library file SYNCDRIV.LIB. This file can be found in the \OS2 directory of disk 1 of the Syncdrive diskette set.

  Ensure that the OS/2 device driver and DLL have been installed properly on the system where the application is to be executed.

All required Syncdrive include files can be found in the \INCLUDE directory of disk 1 of the Syncdrive diskette set.

9.1 Tips and Techniques

Syncdrive data structures must be byte-aligned. They cannot be packed on word or dword boundaries.

The DOS and Windows 3.1 versions of Syncdrive support only large model applications.

When linking, the /NOF option is required. This prevents the optimizing of far calls to near calls for routines that end up in the same code segment.

Use the MPA-200 configuration information and routines for the MPA-300 adapter.

10 Include File Structure

mpa-x00.h - Defines the channel configuration structure and provides function prototypes for Syncdrive API functions.

syncdriv.h - Defines the com_block and BufferQueue structures, data types, global limits, status codes and error codes.

syncos2.h - Defines several items needed to generate Syncdrive applications for OS/2. Within a source code module, this file must be included before any other Syncdrive include files.

snc31dll.h - Defines several items needed to generate Syncdrive applications for Windows 3.1. Within a source code module, this file must be included before any other Syncdrive include files.

snc95dll.h - Defines several items needed to generate Syncdrive applications for Windows 95/98. Within a source code module, this file must be included before any other Syncdrive include files.

11 Example Programs

Numerous small example programs are supplied with Syncdrive. These examples cover the various operating modes of the product. Source code and executable files are provided for each example program. A small selection of makefiles is also provided.

11.1 Source code

The source code for all examples can be found in the \EXAMPLES directory on disk 1 of the Syncdrive diskette set. The examples are text mode programs under DOS, Windows 95/98 and OS/2, and are "QuickWin" text mode programs under Windows 3.1.

11.2 Executable files

The executable programs can be found in the following directories of the Syncdrive diskette set:

✍ DOS	\EXAMPLES\DOS\EXE	(disk 2)
✍ Windows 3.1	\EXAMPLES\WIN31\EXE	(disk 2)
✍ Windows 95/98	\EXAMPLES\WIN95\EXE	(disk 3)
✍ OS/2	\EXAMPLES\OS2\EXE	(disk 3)

11.3 Building the example programs

The Syncdrive example programs have been tested with popular compilers on each supported operating system, but we do not have customer-ready makefiles for all of them. For the makefiles that are supplied, it may be necessary to modify compiler and linker options, drive letters, paths to tools, include files, libraries, etc.

Because the same set of example programs are designed to be used with all MPA-series adapters and all operating systems supported by Syncdrive, it is necessary to define macros when compiling them:

MPA102	Compile for MPA-102 adapter
MPA200	Compile for all other adapters
__OS2__	Compile for OS/2
_WINDOWS	Compile for Windows 3.1/95/98
__WIN95__	Compile for Windows 95/98

11.3.1 DOS

Makefiles are supplied for the Borland C++ compiler (version 3.1) and the Microsoft C compiler (version 6.0). There are also batch files which take a parameter (MPA200 or MPA102) indicating the adapter type and run the appropriate makefile for each application. The files are found on disk 2 of the Syncdrive diskette set in the \EXAMPLES\DOS directory:

DOSAPPSB.BAT	Build all apps with Borland C++ 3.1
BCAPP.MAK	Borland C++ makefile
DOSAPPSM.BAT	Build all apps with Microsoft C 6.0
APP.MAK	Microsoft C makefile

11.3.2 OS/2

A makefile is supplied for the IBM VisualAge C++ (version 3.0) compiler. To use this makefile, run NMAKE from an OS/2 command prompt. The makefile takes a parameter (MPA200 or MPA102) indicating the adapter type. The makefile is located on disk 3 of the Syncdrive diskette set:

All adapters	\EXAMPLES\OS2\OS2APPS.MAK

11.3.3 Windows 3.1

A project file is supplied for the Borland C++ (version 4.5) 16-bit Windows compiler. The project file is located on disk 2 of the Syncdrive diskette set:

MPA-102	\EXAMPLES\WIN31\16APP102.IDE
All others	\EXAMPLES\WIN31\16APP200.IDE

11.3.4 Windows 95/98

A project file is supplied for the Borland C++ (version 5.02) 32-bit Windows compiler. The project file is located on disk 3 of the Syncdrive diskette set:

MPA-102	\EXAMPLES\WIN31\32APP102.IDE
All others	\EXAMPLES\WIN31\32APP200.IDE

11.4 Descriptions of example programs

11.4.1 LPBCKBI, LPBCKBY

This is a loopback program, running in bit-synchronous (LPBCKBI) or byte-synchronous (LPBCKBY) mode. Only one MPA-series adapter is needed. No connector or cable is necessary. Configuration options can be selected on the command line.

11.4.2 FDBCKBI, FDBCKBY

This is a "feedback" program, running in bit-synchronous (FDBCKBI) or byte_synchronous (FDBCKBY) mode. Two MPA-series adapters must be installed in the same system in order to run this program. One adapter must be configured for DTE, and the other for DCE. A one-to-one cable is connected between the two adapters. If an MPA-102 is being used, only one adapter is needed. The addresses and IRQs used are hard-coded (see the source code).

11.4.3 SENDBI

This program transmits single frames. It runs in bit-synchronous mode. It can be used in tandem with RECBI running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.4 RECBI

This program receives single frames. It runs in bit-synchronous mode. It can be used in tandem with SENDBI running on another computer with the two MPA-series adapters (one a DTE, the other a DCE)

connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.5 SENDBY

This program transmits single frames. It runs in byte-synchronous mode. It can be used in tandem with RECBY running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.6 RECBY

This program receives single frames. It runs in byte-synchronous mode. It can be used in tandem with SENDBY running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.7 QSENDBI

This program demonstrates the use of buffer queues for transmitting. It runs in bit-synchronous mode. By default, ring mode is used for the queue. This program can be used in tandem with QRECBI running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. QSENDBI sends frames too quickly for the RECBI program to be used on the other end. Configuration options can be selected on the command line.

11.4.8 QRECBI

This program demonstrates the use of buffer queues for receiving. It runs in bit-synchronous mode. By default, ring mode is used for the queue. This program can be used in tandem with QSENDBI or SENDBI running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.9 THREADRX

This OS/2 program demonstrates Syncdrive's support for multithreaded programs. It spawns a number of threads which take turns setting up bit-synchronous single-frame receive operations. By default, timeouts are enabled. THREADRX can be used in tandem with THREADTX running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

11.4.10THREADTX

This OS/2 program demonstrates Syncdrive's support for multithreaded programs. It spawns a number of threads which take turns setting up bit-synchronous single-frame transmit operations. By default, timeouts will be generated. THREADTX can be used in tandem with THREADRX running on another computer with the two MPA-series adapters (one a DTE, the other a DCE) connected by a one-to-one cable. Configuration options can be selected on the command line.

12 Definitions

DMA (Direct Memory Access) - A method by which the computer hardware can transfer I/O information directly between the port and memory. It is useful because it reduces software overhead thus increasing system throughput. DMA is available to Syncdrive when operating with bit synchronous protocol.

SCC (Serial Communications Controller) - The integrated circuit that handles the bulk of the hardware functions for the serial channel. Typically, the SCC will either be an AM85C30-10 or Z85230-10 on Quatech MPA-series adapters.

13 MPA-Series Adapter Clocking Options

The various MPA-series adapters have slightly different options for routing clock signals to and from the connector. Single-channel MPA-series adapters use a portion of the "B" channel of the SCC to support extra clocking options.

Listed below is a description of the options available for each type of MPA-series adapter supported by Syncdrive. The clock configuration is determined by the *clock_source* variable in the channel configuration structure. Please refer to the connector descriptions in the Hardware Guide for the MPA-series adapter for details on the connector pinouts and signal names.

13.1 MPA-100 DTE

To receive the transmit clock on TXCLK(DCE), set *clock_source* bit D2 to 0 and select the TRxC source. To self-source the transmit clock, set *clock_source* bit D2 to 1 and select either the BRG or DPLL source. The transmit clock is always output on TXCLK(DTE).

To receive the receive clock on RXCLK(DCE), set *clock_source* bit D5 to 0 and select the RTxC source. To self-source the receive clock, set *clock_source* bit D5 to 1 and select either the BRG or DPLL source. The DTE cannot transmit its receive clock.

13.2 MPA-100 DCE

The transmit clock is always self-sourced. Select either the BRG or DPLL source. Set *clock_source* bit D2 to 1 to output the transmit clock on RXCLK(DCE). Setting *clock_source* bit D2 to 0 has no effect. The DCE cannot receive its transmit clock.

To receive the receive clock on RXCLK(DTE), set *clock_source* bit D5 to 0 and select the RTxC source. To self-source the receive clock, set *clock_source* bit D5 to 1 and select either the BRG or DPLL source. The receive clock is always output on TXCLK(DCE).

13.3 MPA-200/300 DTE, MPA-2000/3000 DTE

To receive the transmit clock on RTCLK, set *clock_source* bit D2 to 0 and select the TRxC source. To self-source the transmit clock and output the transmit clock on TTCLK, set *clock_source* bit D2 to 1 and select either the BRG or DPLL source.

To receive the receive clock on RRCLK, set *clock_source* bit D5 to 0 and select the RTxC source. To self-source the receive clock, set *clock_source* bit D5 to 1 and select either the BRG or DPLL source. The DTE cannot transmit its receive clock.

13.4 MPA-200/300 DCE, MPA-2000/3000 DCE

The transmit clock is always self-sourced. Select either the BRG or DPLL source. To output the transmit clock on RRCLK, set *clock_source* bit D2 to 1. Setting *clock_source* bit D2 to 0 has no effect, as the DCE cannot receive its transmit clock.

To receive the receive clock on TTCLK, set *clock_source* bit D5 to 0 and select the RTxC source. To self-source the receive clock and output it on RTCLK, set *clock_source* bit D5 to 1 and select either the BRG or DPLL source.

13.5 MPA-102 DTE

To receive the transmit clock on TXCLK(DCE), set *clock_source* bit D2 to 0 and select the TRxC source. To self-source the transmit clock, set *clock_source* bit D2 to 1 and select either the BRG or DPLL source. The transmit clock is always output on TXCLK(DTE).

The receive clock is always received on TXCLK(DCE). The *clock_source* bit D5 has no effect. Choose the RTxC source. The DTE cannot transmit its receive clock.

13.6 MPA-102 DCE

The transmit clock is always output on both TXCLK(DCE) and RXCLK(DCE). The *clock_source* bit D2 should be set to 1. Choose either the BRG or DPLL source. The DCE cannot receive its transmit clock.

The receive clock is always received on TXCLK(DTE). The *clock_source* bit D5 has no effect. Choose the RTxC source.

13.7 MPAP-100 (only available as a DTE)

Refer to MPA-100 DTE. The receive clock is always output on RXCLK(DTE).

13.8 MPAP-200/300 (only available as a DTE)

Refer to MPA-200/300 DTE.

14 Troubleshooting

14.1 Verify that the hardware is configured properly

A good first step in solving problems is establishing that there are no hardware conflicts. One way to do this is to try known-good Syncdrive applications on the hardware. The compiled Syncdrive example programs are good tools to use for this purpose. All of the compiled programs have been verified to run in a known-good system before they are shipped.

The best place to start is with the LPBCKBI.EXE program. This program requires no cables to be connected. By default, LPBCKBI will run using base address 0x300, IRQ5, DMA3 for transmit and DMA1 for receive. It can accept as command line input whatever resource settings are being used on the installed MPA-series adapter. Run the program with a /? parameter on the command line for a short help screen.

LPBCKBI will run continuous frames in loopback mode. As it runs, it prints each frame to the screen. If LPBCKBI is able to run successfully, then it verifies that the hardware configuration is good. If LPBCKBI crashes or locks up, it indicates that the hardware is not configured properly. It may be necessary to change some of the hardware settings to avoid conflicts with other devices in the system.

14.2 Syncdrive is not PCMCIA-aware

Syncdrive currently is not plug-and-play capable. When using Syncdrive with PCMCIA cards, it is necessary to obtain the base address and IRQ assigned to the card by the PCMCIA Card Services and provide those values in the channel configuration array.

For DOS, Windows 3.1 and OS/2, the client driver supplied with the card should be used to configure the card at the desired base address and IRQ. Use these settings with the Syncdrive application.

Under Windows 95/98, the card is automatically configured. To find the settings, click the right mouse button on the My Computer icon and select Properties. Select the Device Manager tab and double-click the card's entry under the "Synchronous Communication" section. Select the Resources tab to see the card's base address and IRQ. Use these settings

with the Syncdrive application. Windows 95/98 may allow changes to the settings if the "Use Automatic Settings" box is unchecked.

14.3 Check the Syncdrive configuration

If the hardware is verified to be good, then the problem must be somewhere in the software. The first place to check is the channel configuration structure. Double check that all variables in this structure are set properly. If lockups or crashes are a problem, particularly check the hardware settings such as *base_address, tx_interrupt, rx_interrupt, tx_dma_channel,* and *rx_dma_channel*.

14.4 Check the clock sourcing

If transmit or receive operations just don't seem to work right, double check the *clock_source* variable in the channel configuration structure. In conjunction with the hardware manual for the MPA-series adapter, make sure that any clock signals on the cable are being routed to the proper SCC inputs and outputs, and that the modes in *clock_source* are set properly.

14.5 Know the speed limits

There are limits on how fast the MPA-series adapters can operate. Most of the limitations are the result of the ISA bus itself. When using DMA, the top data rate is usually around 1.2 Mbps. When not using DMA, the top data rate is usually around 130-150 kbps. Using an optional Zilog Z85230 SCC with its FIFOs enabled can help increase the achievable data rate when not using DMA.

14.6 Know the possible data rates

Each MPA-series adapter's hardware manual contains an equation used in programming the SCC's baud rate generator. If you are having trouble getting a particular data rate to work, check it with this equation to make sure that the Baud_Const divisor is an integer value. If it is not an integer value, Syncdrive will round the data rate to the nearest rate that will produce an integer divisor.

In the baud rate divisor equation, Clock_Frequency and "Clock_Mode" are the respective values from the *clock_rate* and *clock_source* (bits D7 and D6) variables in the channel configuration structure. Notice that the equation shows that as baud rate increases, the steps between baud rate selections get larger. Also note that the maximum available baud rate on the SCC is 1/4 of the input clock frequency. Unfortunately, because the ISA bus cannot provide service to the adapter quickly enough, rates this high cannot actually be achieved.

If you need a baud rate that cannot be achieved with the standard clock frequency, contact Quatech Technical Support. It may be possible to use a custom clock oscillator to change the available baud rate selections.

15 Error Codes

The definitive source for error codes is the SYNCDRIV.H include file. The error codes for Release 4.00 are duplicated here for convenience.

0x0000 Function successfully completed
0x0xxx User error (see below)
0x1xxx Internal Syncdrive error (report to Quatech)

```
#define SYNC_SUCCESS                            0x0000  // Function complete
#define BAD_CFG_POINTER                         0x0001  // Invalid channel_cfg pointer
#define BAD_SIGNATURE                           0x0002  // Invalid signature (channel_cfg)
#define BAD_BOARD_ID                            0x0003  // Invalid board type (channel_cfg)
#define NO_BISYNC_MASK                          0x0004  // Not running BISYNC mode
#define CHANNEL_ALREADY_CONFIGURED      0x0005  // Invalid repeat config_mpaxxxx call
#define BAD_MASK_TYPE                           0x0006  // Rx char type out of range
#define CANNOT_SET_STATUS                       0x0007
#define CANNOT_SET_BAUD                         0x0008
#define CANNOT_RESET_RX                         0x0009
#define BAD_QUEUE_MODE                          0x000A  // Invalid BufferQueue.QueueMode
#define CANNOT_ALLOCATE_DMA_BUFFER      0x000B
#define CANNOT_FREE_DMA_BUFFERS                 0x000C
#define TOO_MANY_DMA_BUFFERS                    0x000D  // DMA buffer count limit exceeded
#define BOARD_NUMBER_OUT_OF_RANGE               0x000F  // Invalid board_number (channel_cfg)
#define BAD_DMA_CHANNEL_NUMBER                  0x0010  // DMA 1, 2, or 3 are valid
#define TOO_MANY_COMBLOCKS                      0x0011  // See GLOBAL_MAXCOMBLOCKS
#define CANNOT_ABORT_TX                         0x0012
#define QUEUE_ALREADY_REGISTERED                0x0013  // Free the registered queue first
#define QUEUE_NOT_REGISTERED                    0x0014  // Register a queue first
#define QUEUE_ALREADY_RUNNING                   0x0015  // Abort the Rx or Tx operation first
#define CHANNEL_NOT_OPEN                        0x0016  // Call config_mpaXXXX first
#define COMBLOCK_TOO_LARGE                      0x0017  // Exceeds 64kB, including overhead
#define NULL_BUFFER_LENGTH                      0x0018  // Comblock buffer_length was 0.
#define VERSION_MISMATCH                        0x0019  // DLL, driver versions don't match
#define BAD_IRQ_NUMBER                          0x0020  // Invalid IRQ or IRQ in use
#define INVALID_COMMAND                         0x0030  // Command number out of range
#define BAD_BAUD_RATE                           0x0031  // Maximum = (0.25)(clock rate)
#define QUEUE_DMA_SETUP_ERROR                   0x0032  // BufferArray[] contains non-DMA
buffer.
#define DMA_SETUP_ERROR                         0x0033  // Use sync_alloc_dma_buffer with
DMA.
#define DRIVER_BUSY                             0x0082  // Still processing an earlier frame

#define FATAL_CANNOT_GET_SEMAPHORE              0x1001
#define FATAL_CANNOT_RELEASE_SEMAPHORE          0x1002
#define FATAL_CANNOT_CREATE_SEMAPHORE           0x1003
#define FATAL_CANNOT_DESTROY_SEMAPHORE          0x1004
#define FATAL_CANNOT_OPEN_CHANNEL       0x1005
#define FATAL_CANNOT_CLOSE_CHANNEL              0x1006
#define FATAL_CANNOT_SET_SYSINFO                0x1007
#define FATAL_CANNOT_MALLOC_OBJECT              0x1008
#define FATAL_CANNOT_MALLOC_MASK                0x1009
#define FATAL_CANNOT_MALLOC_QUEUE       0x100A
#define FATAL_CANNOT_STORE_CONFIG_ARRAY         0x100B
#define FATAL_CANNOT_LOCK_MEMORY        0x100C
#define FATAL_GDT_MAPPING_FAILED                0x100D
#define FATAL_CANNOT_UNLOCK_MEMORY              0x100E
```

```
#define FATAL_CANNOT_GET_GLOBAL_MAP          0x100F
#define FATAL_CANNOT_FREE_GLOBAL_MAP         0x1010
#define FATAL_CANNOT_FIND_DRIVER             0x1011
```