SECOND EDITION

MICROPROCESSOR SYSTEMS DESIGN

68000 HARDWARE, SOFTWARE, AND INTERFACING

ALAN CLEMENTS
TEESSIDE POLYTECHNIC

PWS-KENT PUBLISHING COMPANY
BOSTON

CHAPTER NINE

THE SERIAL INPUT/ OUTPUT INTERFACE

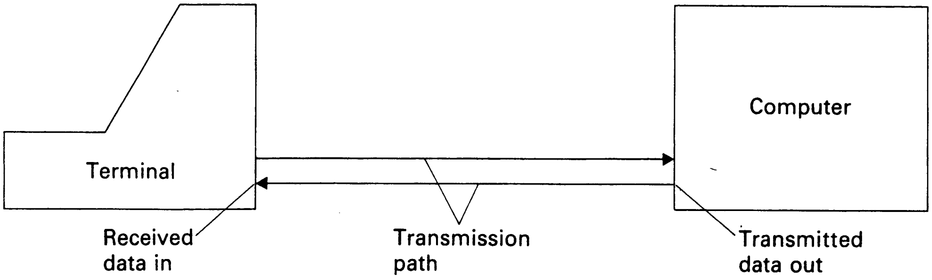
The vast majority of general-purpose microcomputers, except some entirely self-contained portable models, use a serial interface to communicate with remote peripherals such as CRT terminals. The serial interface, which moves information from point to point one bit at a time, is generally preferred to the parallel interface, which is able to move a group of bits simultaneously. This preference is not due to the high performance of a serial data link but to its low cost, simplicity, and ease of use. In this chapter we first describe how information is transmitted serially and then examine a typical parallel-to-serial and serial-to-parallel chip that forms the interface between a microprocessor and a serial data link. Because a serial data link can operate in one of two modes, asynchronous or synchronous, a separate section is devoted to each mode. We also take a brief look at some of the standards for the transmission of serial data. The chapter ends with the description of a suitable serial interface for a 68000-based system. Throughout this chapter, the word *character* refers to the basic unit of information transmitted over a data link. The term character has been chosen because many data links transmit information in the form of text, so that the unit of transmitted information corresponds to a printed character.

Figure 9.1 illustrates the basic serial data link between a computer and a CRT terminal. A CRT terminal requires a two-way data link because information from the keyboard is transmitted to the computer and information from the computer is transmitted to the screen. Note that the *transmitted* data from the computer becomes the *received* data at the CRT terminal. Although this statement is an elementary and self-evident observation, confusion between transmitted and received data is a common source of error in the linking of computers and terminals.

A more detailed arrangement of a serial data link in terms of its functional components is given in figure 9.2. The heart of the data link is the box labeled "serial interface," which translates data between the form in which it is stored within the computer and the form in which it is transmitted over the data link. The conversion of data between parallel and serial form is often performed by a single LSI device called an asynchronous communications interface adaptor (ACIA).

The line drivers in figure 9.2 have the function of translating the TTL level signals processed by the ACIA into a suitable form for sending over the transmission path. The transmission path itself is normally a twisted pair of conductors, which accounts for its very low cost. Some systems employ more esoteric transmission

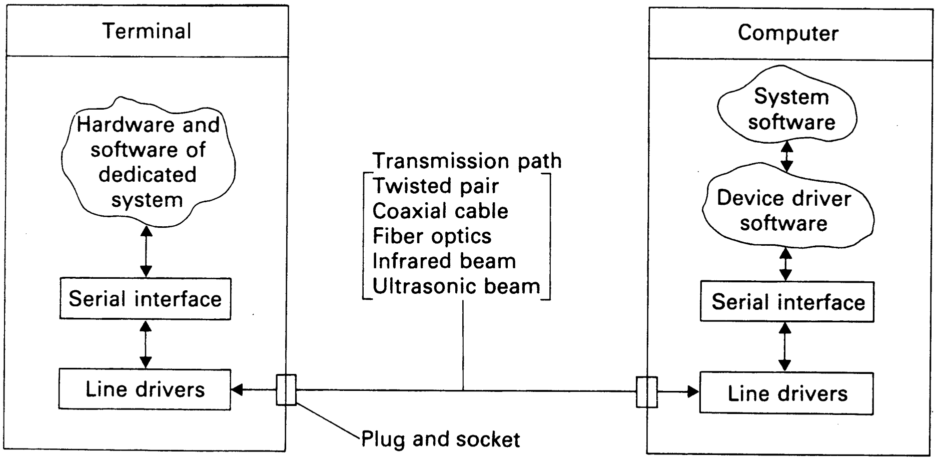
FIGURE 9.1 Serial data link



paths such as fiber optics or infrared (IR) links. The connection between the line drivers and transmission path is labeled *plug and socket* in figure 9.2 to emphasize that such mundane things as plugs become very important if interchangeability is required. International specifications cover this situation and other aspects of the data link.

The two items at the computer end of the data link enclosed in "clouds" in figure 9.2 represent the software components of the data link. The lower cloud contains the software that directly controls the serial interface itself by performing operations such as transmitting a single character or receiving a character and checking it for certain types of error. On top of this software sits the application-level software, which uses the primitive operations executed by the lower-level software to carry out actions such as listing a file on the screen.

FIGURE 9.2 Functional units of a serial data link



9.1 ASYNCHRONOUS SERIAL DATA TRANSMISSION

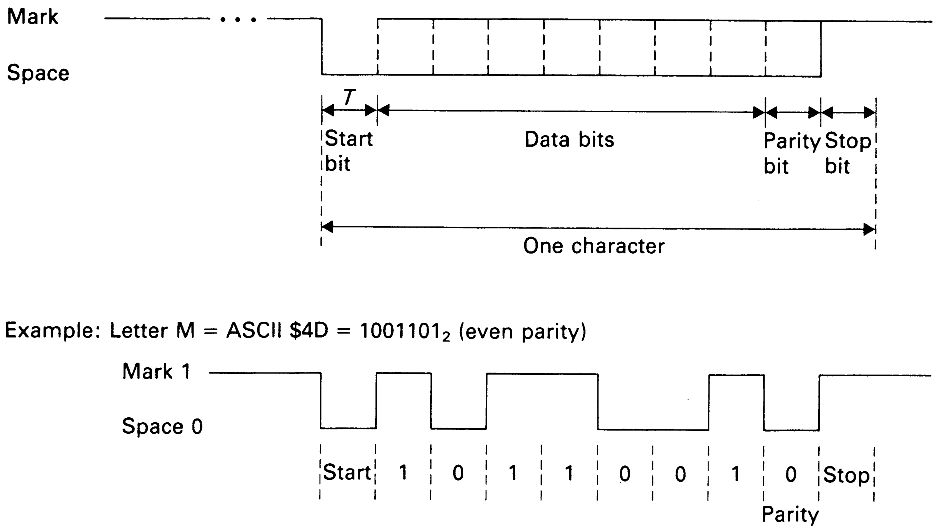
By far the most popular serial interface between a computer and its CRT terminal is the asynchronous serial interface. This interface is so called because the transmitted data and the received data are not synchronized over any extended period and therefore no special means of synchronizing the clocks at the transmitter and receiver is necessary. In fact, the asynchronous serial data link is a very old form of data transmission system and has its origin in the era of the teleprinter.

Serial data transmission systems have been around for a long time and are found in the telephone (human speech), Morse code, semaphore, and even the smoke signals once used by native Americans. The fundamental problem encountered by all serial data transmission systems is how to split the incoming data stream into individual units (i.e., bits) and how to group these units into characters. For example, in Morse code the dots and dashes of a character are separated by an intersymbol space, while the individual characters are separated by an intercharacter space, which is three times the duration of an intersymbol space.

First we examine how the data stream is divided into individual bits and the bits grouped into characters in an asynchronous serial data link. The key to the operation of this type of link is both simple and ingenious. Figure 9.3 gives the format of data transmitted over such a link.

An asynchronous serial data link is said to be character oriented, as information is transmitted in the form of groups of bits called characters. These characters are invariably units comprising 7 or 8 bits of "information" plus 2 to 4 control bits and

FIGURE 9.3 Format of asynchronous serial data



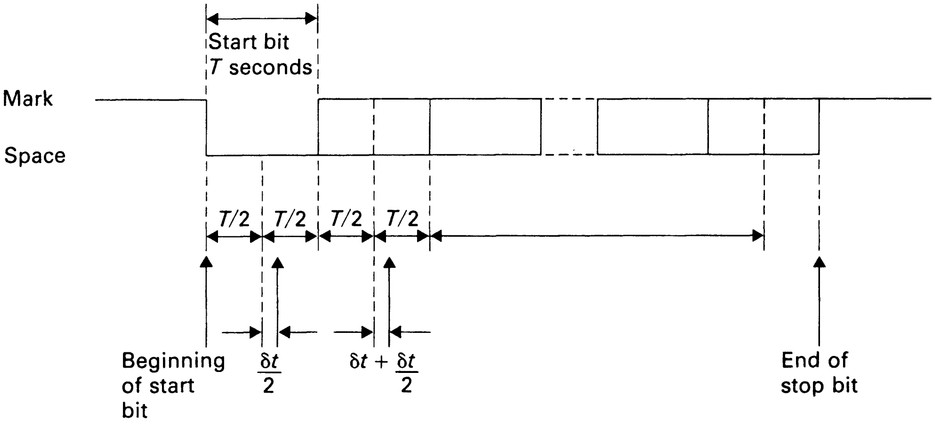
frequently correspond to ASCII-encoded characters. Initially, when no information is being transmitted, the line is in an idle state. Traditionally, the idle state is referred to as the *mark level.* By convention this corresponds to a logical 1 level.

When the transmitter wishes to send data, it first places the line in a *space* level (i.e., the complement of a mark) for one element period. This element is called the start bit and has a duration of T seconds. The transmitter then sends the character, 1 bit at a time, by placing each successive bit on the line for a duration of T seconds, until all bits have been transmitted. Then a single parity bit is calculated by the transmitter and sent after the data bits. Finally, the transmitter sends a stop bit at a mark level (i.e., the same level as the idle state) for one or two bit periods. Now the transmitter may send another character whenever it wishes. The only purpose of the stop bit is to provide a rest period for the receiver between consecutive characters. This bit is a relic of the days of electromechanical receivers and is not now strictly required for technical reasons, existing only for the purpose of compatibility with older equipment.

As the data wordlength may be 7 or 8 bits with odd, even, or no parity bits, plus either 1 or 2 stop bits, a total of 12 different possible formats can be used for serial data transmission—and this is before we consider that there are about seven commonly used values of T, the element duration. Connecting one serial link with another may therefore be difficult because so many options are available.

At the receiving end of an asynchronous serial data link, the receiver continually monitors the line looking for a start bit. Once the start bit has been detected, the receiver waits until the end of the start bit and then samples the next N bits at their centers, using a clock generated locally by the receiver. As each incoming bit is sampled, it is used to construct a new character. When the received character has been assembled, its parity is calculated and compared with the received parity bit following the character. If they are not equal, a parity error flag is set to indicate a transmission error.

FIGURE 9.4 Effect of unsynchronized transmitter and receiver clocks



NOTE: Vertical lines with arrows indicate the points at which the received data is sampled.

The most critical aspect of the system is the receiver timing. The falling edge of the start bit triggers the receiver's local clock, which samples each incoming bit at its nominal center. Suppose the receiver clock waits $T/2$ seconds from the falling edge of a start bit and samples the incoming data every T seconds thereafter until the stop bit has been sampled. Figure 9.4 shows this situation. As the receiver's clock is not synchronized with the transmitter clock, the sampling is not exact.

Let us assume that the receiver clock is running slow, so that a sample is taken every $T + \delta t$ seconds. The first bit of the data is sampled at $(T + \delta t)/2 + (T + \delta t)$ seconds after the falling edge of the start bit. The stop bit is sampled at time $(T + \delta t)/2 + N(T + \delta t)$, where N is the number of bits in the character following the start bit. The total accumulated error in sampling the stop bit is therefore $(T + \delta t)/2 + N(T + \delta t) - (T/2 + NT)$, or $(2N + 1)\delta t/2$ seconds. For correct operation, the stop bit must be sampled within $T/2$ seconds of its center, so that:

$$\frac{T}{2} > \frac{(2N + 1)\delta t}{2}$$

or

$$\frac{\delta t}{T} < \frac{1}{2N + 1}$$

or

$$\frac{\delta t}{T} < \frac{100}{2N + 1} \quad \text{as a percentage}$$

If $N = 9$ for a 7-bit character + parity bit + 1 stop bit, the maximum permissible error is $100/19 = 5$ percent. Fortunately, almost all clocks are now crystal controlled, and the error between transmitter and receiver clocks is likely to be a tiny fraction of 1 percent.

The most obvious disadvantage of asynchronous data transmission is the need for a start, parity, and stop bit for each transmitted character. If 7-bit characters are used, the overall efficiency is only $7/(7 + 3) \times 100 = 70$ percent. A less obvious disadvantage is due to the character-oriented nature of the data link. Whenever the data link connects a CRT terminal to a computer, few problems arise, as the terminal is itself character oriented. However, if the data link is being used to, say, dump binary data to a magnetic tape, problems arise. If the data are arranged as 8-bit bytes with all 256 possible values corresponding to valid data elements, it is difficult (but not impossible) to embed control characters (e.g., tape start or stop) within the data stream because the same character must be used both as pure data (i.e., part of the message) and for control purposes.

If 7-bit characters are used, pure binary data cannot be transmitted in the form of one character per byte. Two characters are needed to record each byte and this condition is clearly inefficient. We will see later how synchronous serial data links overcome this problem.

We have now described how information can be transmitted serially in the form of 7- or 8-bit characters. The next step is to show how these characters are encoded.

ASCII Code

Although computing generally suffers from a lack of standardization, the ASCII code is one of the few exceptions. Many microcomputers employ the ASCII code to represent information in character form internally, and for the exchange of information between themselves and CRT terminals. The ASCII code, or American Standard Code for Information Interchange, is one of several codes used to represent alphanumeric characters. As long ago as the 1920s, the Baudot or Murray code was designed for the teleprinter. This code, still used by the international telex service, represents characters by 5 bits. As this system provides only 2^5 unique values, one of the 32 possible values acts as a shift, affecting the meaning of the following characters. The effective number of characters available is thereby increased.

The ASCII code employs 7 bits to give a total of 128 unique values. These bits are sufficient to provide a full 96-character upper- and lowercase printing set, together with 32 characters to control the operation of the data link and the terminal itself. The ASCII code has now been adopted universally, and is almost identical to the International Standards Organization ISO-7 code.

Had the ASCII code been developed today, it would almost certainly be an 8-bit code. Unfortunately, the ASCII character set does not include "national" char-

TABLE 9.1 ASCII code

	$b_6b_5b_4$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
$b_3b_2b_1b_0$		**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0	0000	NUL	DLC	SP	0	@	P	'	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LT	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	—	=	M	]	m	}
E	1110	SO	RS	.	>	N	Λ	n	~
F	1111	SI	VS	/	?	O	_	o	DEL

acters such as the German umlaut or the French accents. Moreover, a graphical character set similar to that used by Teletex would have been very helpful. However, microcomputer manufacturers have tended to design their own graphics codes, leading to incompatibiliy.

Table 9.1 presents the ASCII code. The binary value of a character is obtained by reading the three most significant bits at the top of the column in which the character occurs and then taking the four least significant bits from its row. For example, the character *m* is in the column headed 110 and the row headed 1101; therefore, the binary code for *m* is 110 1101 (or $6D in hexadecimal).

9.2 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTOR (ACIA)

One of the first general-purpose interface devices produced by the semiconductor manufacturers was the asynchronous communications interface adaptor, or ACIA. The ACIA relieves the system software of all the basic tasks involved in converting data between serial and parallel forms; that is, the ACIA contains almost all the logic necessary to provide an asynchronous data link between a computer and an external system.

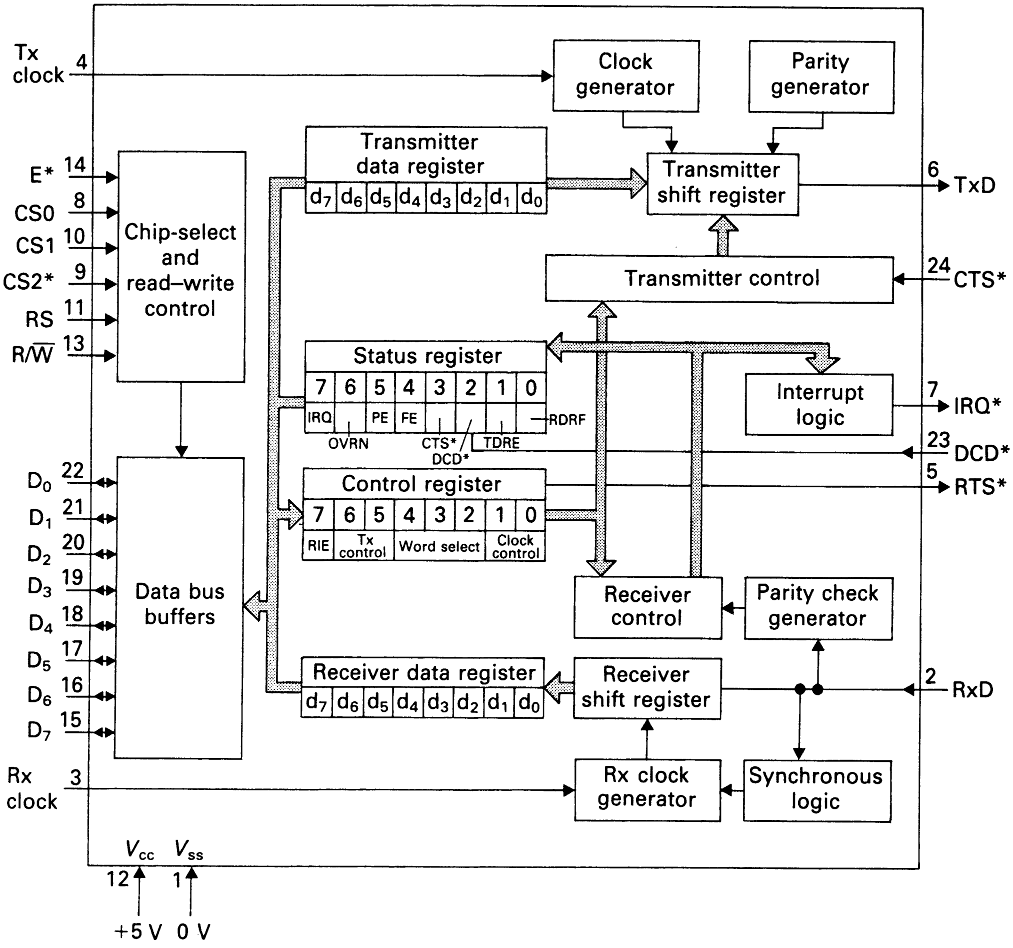
One of the earliest and still popular ACIAs is the 6850 illustrated in figure 9.5. This particular ACIA will be described because it is much easier to understand than some of the newer ACIAs and is still widely used in microcomputers. Once the reader understands how the 6850 ACIA operates, he or she can read the data sheet of any other ACIA. Like any other digital device, the 6850 has a hardware model, a software model, and a functional model. We look at the hardware model first. Figure 9.6 gives the hardware model of the 6850 together with its timing diagram. From the designer's point of view, the 6850's hardware can be subdivided into three sections: the CPU side, the transmitter side, and the receiver side.

CPU Side

As far as the CPU is concerned, the 6850 behaves almost exactly like a static read/write memory; figure 9.6 shows the read and write cycles on the same diagram. However, one important difference exists between the 6850 and conventional RAM. The 6850's memory accesses are synchronized to an external E or enable clock. In 6800- or 6809-based systems, this situation presents no problem as the processor itself is also synchronized to the E (or $\phi 2$ in 6800 terminology) clock that it provides. In the case of the 68000, the ACIA must be interfaced either by using VPA* and VMA* or asynchronously by means of additional logic.

The ACIA is a byte-oriented device and can be interfaced to D_{00} through D_{07} and strobed by LDS*, or to D_{08} through D_{15} and strobed by UDS*. The ACIA has a

FIGURE 9.5 The 6850 ACIA



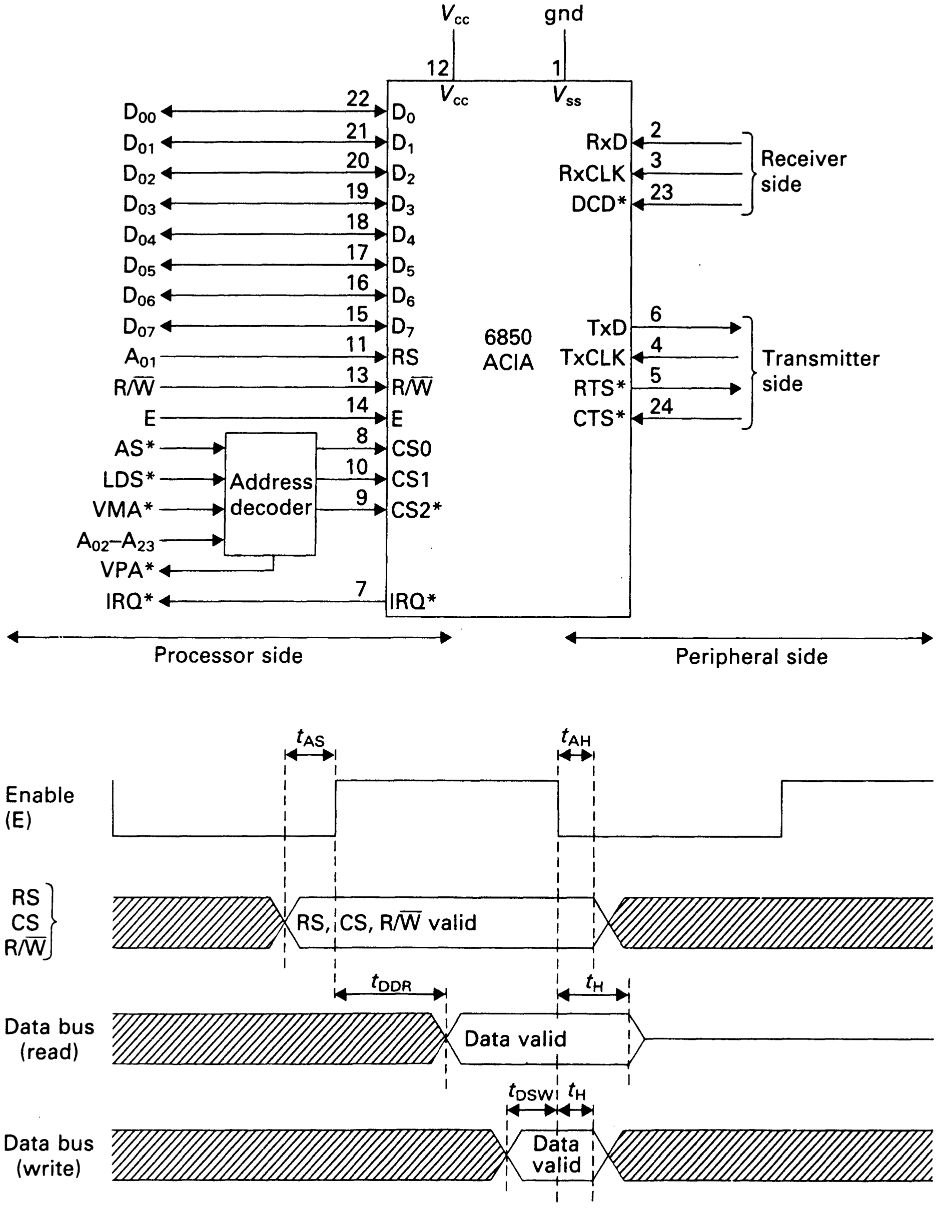
single register select line, RS, that determines the internal location (i.e., register) addressed by the processor. Typically, RS is connected to the processor's A_{01} address output, so that the lower location is selected by address X and the upper, by address X + 2.

Three chip-select inputs are provided, two of which are active-high and one, active-low. This spectacular display of overkill comes from the days when address decoders were relatively expensive and memories small. By using the chip selects alone, you can achieve partial address decoding without any additional components. In many modern systems just one of the chip-select inputs takes part in the address decoding process. The remaining two are permanently enabled. This situation is unfortunate, as the other two pins could have provided the ACIA with additional features—such as a RESET* input or an on-chip clock.

The 6850 has an interrupt request output, IRQ*, that can be connected to any of the 68000's seven levels of interrupt request input. As the 6850 does not support vectored interrupts, autovectored interrupts must be used in the way described in chapter 6.

Unusually, the 6850 does not have a RESET* input because there were not

FIGURE 9.6 Hardware model of the ACIA and its timing diagram



enough pins to provide the function and the manufacturer felt that RESET* was the most dispensable of functions. When power is first applied, some sections of the ACIA are reset automatically by an internal power-on-reset circuit. Afterwards, a secondary reset by software is performed, as we shall describe later.

The CPU side of the 6850 has a clock input labeled E (i.e., ENABLE). As with other 6800-series peripherals, the E input must be both free running and synchronized to read/write accesses between the ACIA and the processor. The simplest interface between the ACIA and a 68000 processor is given in figure 9.7. This circuit is entirely conventional and makes use of the CPU's synchronous bus control signals.

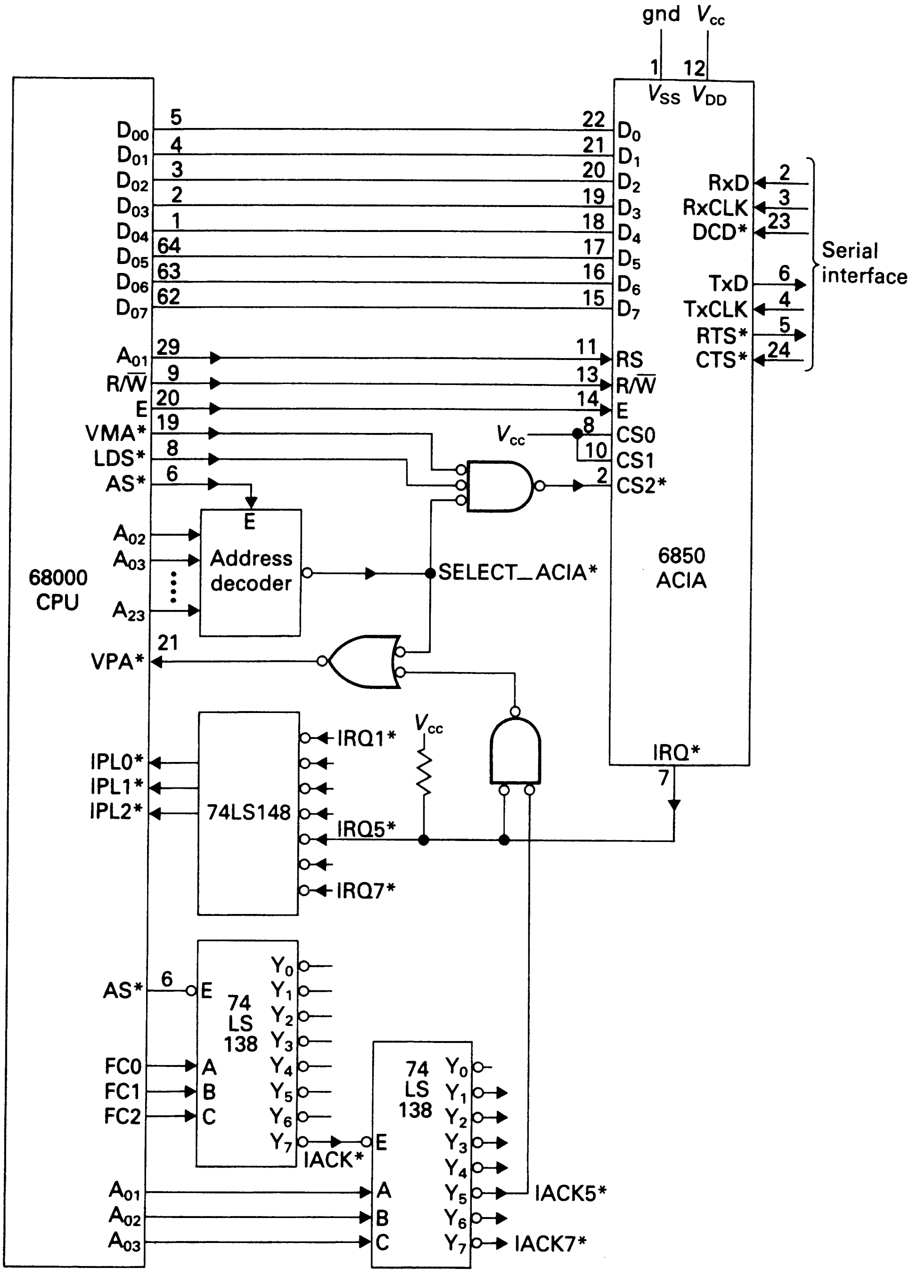
The lower byte of the 68000's data bus is connected to the ACIA's data input/output pins, D_0 to D_7, which locate all the ACIA's registers in the lower half of words at odd addresses. Remember that the 68000 address space is arranged so that lower-order bits (D_{00} to D_{07}) have odd addresses and higher-order bits (D_{08} to D_{15}) have even addresses. Whenever the 68000 addresses the ACIA, the address decoder detects the access and forces SELECT_ACIA* low. This signal drives the 68000's VPA* low via an OR gate, signaling that a synchronous bus cycle is to begin. The CPU then forces VMA* low and the ACIA is selected by SELECT_ACIA*, VMA*, and LDS* all being low simultaneously. During this access, $R/\overline{W}$ from the CPU determines the direction of data transfer and A_{01} the location of the internal register selected in the ACIA.

The lower portion of figure 9.7 is intended to show how the ACIA is operated in the autovectored interrupt mode. When the ACIA forces its IRQ* line low, a level 5 interrupt is signaled to the CPU. Assuming this level is enabled, IACK5* from the decoder goes low and is then ANDed with IRQ* from the ACIA and connected to VPA* via an OR gate. The purpose of ANDing IACK5* with IRQ* is to permit an interrupt acknowledge to the 68000 (via VPA*) only when the ACIA is putting out an interrupt request while an interrupt acknowledge at the appropriate level is being indicated by the CPU.

Receiver and Transmitter Sides of the ACIA

One of the great advantages of peripherals such as the 6850 ACIA is that they isolate the CPU from the outside world both physically and logically. The *physical isolation* means that the engineer who is connecting a peripheral device to a microprocessor system does not have to worry about the electrical and timing requirements of the CPU itself. In other words, all the engineer needs to understand about the ACIA is the nature of its transmitter-side and receiver-side interfaces. Similarly, the peripheral performs a *logical isolation* by hiding the details of information transfer across it; for example, the operation of transmitting a character from an ACIA is carried out by the instruction MOVE.B D0,ACIA_DATA, where register D0 contains the character to be transmitted and ACIA_DATA is the address of the data register in the ACIA. All the actions necessary to actually serialize the data and append start, parity, and stop bits are carried out automatically (i.e., invisibly) by the ACIA.

FIGURE 9.7 Interface between a 6850 ACIA and a 68000 CPU



Here, only the essential details of the ACIA's transmitter and receiver sides are presented, because the way in which they function is described more fully when we come to the logical organization of the 6850. The peripheral-side interface of the 6850 is divided into two entirely separate groups—the receiver group, which forms the interface between the ACIA and a source of incoming data, and the transmitter group, which forms the interface between the ACIA and the destination for outgoing data. *Incoming* and *outgoing* are used with respect to the ACIA. The nature of these signals is strongly affected by one particular role of the ACIA—its role as an interface between a computer and the public switched telephone network via a modem.

Receiver Side

Incoming data to the ACIA is handled by three pins: RxD, RxCLK, DCD*. Like all other inputs and outputs to the ACIA, these are TTL-level compatible signals. The RxD (receiver data input) pin receives serial data from the transmission path to which the ACIA is connected. The idle (mark) state at this pin is a TTL logical 1 level. A receiver clock is provided at the RxCLK (receiver clock) input pin by the systems designer. The RxCLK clock must be either the same, 16, or 64 times the rate at which bits are received at the data input terminal. Many modern ACIAs include on-chip receiver and transmitter clocks, relieving the system designer of the necessity of providing an additional external oscillator.

The third and last component of the receiver group is an active-low DCD* (data carrier detect) input. DCD* is intended for use in conjunction with a modem and, when low, indicates to the ACIA that the incoming data is valid. When inactive-high, DCD* indicates that the incoming data might be erroneous. This situation may arise if the level (i.e., signal strength) of the data received at the end of a telephone line drops below a predetermined value or the connection itself is broken.

Transmitter Side

The transmitter side of the ACIA comprises four pins: TxCLK, TxD, RTS*, and CTS*. The transmitter clock input (TxCLK) provides a timing signal from which the ACIA derives the timing of the transmitted signal elements. In most applications of the ACIA, the transmitter and receiver clocks are connected together and a common oscillator used for both transmitter and receiver sides of the ACIA. Serial data is transmitted from the TxD (transmit data) pin of the ACIA, with a logical one level representing the idle (mark) state.

An active-low request to send (RTS*) output indicates that the ACIA is ready to transmit information. This output is set or cleared under software control and can be used to switch on any equipment needed to transmit the serial data over some data link. Some use it to switch on a cassette recorder when the ACIA is interfaced to a magnetic tape recording system.

An active-low clear to send (CTS*) input indicates to the transmitter side of the ACIA that the external equipment used to transmit the serial data is ready. When negated, this input inhibits the transmission of data. CTS* is a modem signal that indicates that the transmitter carrier is present and that transmission may go ahead.

Operation of the 6850 ACIA

The software model of the 6850 has four user-accessible registers, as defined in table 9.2. These registers are a transmit data register (TDR), a receive data register (RDR), a system control register (CR), and a system status register (SR). As there are four registers and yet the ACIA has only a single register-select input, RS, a way must be found to distinguish betweeп registers. The ACIA uses the R/$\overline{W}$ input to make this distinction. Two registers are read-only (i.e., RDR, SR) and two are write-only (TDR, CR). Although a perfectly logical, indeed an elegant, thing to do, I do not like it. I am perfectly happy to accept read-only registers, but I am suspicious of the write-only variety because the contents of a write-only register are impossible to verify. Suppose a program with a bug executed an unintended write to a write-only register. The change cannot be detected by reading back the contents of the register.

Table 9.2 also gives the address of each register, assuming that the base address of the ACIA is $00 E001 and that it is selected by LDS*. The purpose of this exercise is twofold: it shows that the address of the lower-order byte is odd and that the pairs of read-only and write-only registers are separated by two (i.e., $00 E001 and $00 E003).

Control Register

Because the ACIA is a versatile device and can be operated in any of several different modes, the control register permits the programmer to define its operational characteristics. This job can even be done dynamically if the need ever arises. Table 9.3 shows how the 8 bits of the control register are grouped into four logical fields.

Bits CR0 and CR1 determine the ratio between the transmitted or received bit rates and the transmitter and receiver clocks, respectively. The clocks operate at the same, 16, or 64 times the data rate. Most applications of the 6850 employ a receiver/transmitter clock at 16 times the data rate with CR1 = 0 and CR0 = 1. Setting CR1 = CR2 = 1 is a special case and serves as a software reset of the ACIA. A software reset clears all internal status bits, with the exception of CTS* and DCD*. A software reset to the 6850 is invariably carried out during the initialization phase of the host processor's reset procedures.

The *word-select* field, bits CR2, CR3, and CR4, determines the format of the

TABLE 9.2 Register-selection scheme of the 6850 ACIA

ADDRESS	RS	R/$\overline{W}$	REGISTER TYPE	REGISTER FUNCTION	
00 E001	0	0	Write only	Control register	(CR)
00 E001	0	1	Read only	Status register	(SR)
00 E003	1	0	Write only	Transmit data register	(TDR)
00 E003	1	1	Read only	Receive data register	(RDR)

NOTE: Base address of ACIA = $00 E001.

TABLE 9.3 Structure of the ACIA's control register

BIT	CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0
Function	Receiver interrupt enable	Transmitter control		Word select			Counter division	

CR1	CR0	DIVISION RATIO
0	0	1
0	1	16
1	0	64
1	1	Master reset

CR4	CR3	CR2	WORD SELECT			
			DATA WORD LENGTH	PARITY	STOP BITS	TOTAL BITS
0	0	0	7	Even	2	11
0	0	1	7	Odd	2	11
0	1	0	7	Even	1	10
0	1	1	7	Odd	1	10
1	0	0	8	None	2	11
1	0	1	8	None	1	10
1	1	0	8	Even	1	11
1	1	1	8	Odd	1	11

CR6	CR5	TRANSMITTER CONTROL	
		RTS*	TRANSMITTER INTERRUPT
0	0	Low (0)	Disabled
0	1	Low (0)	Enabled
1	0	High (1)	Disabled
1	1	Low (0)	Disabled and break

CR7	RECEIVER INTERRUPT ENABLE
0	Receiver may not interrupt
1	Receiver may interrupt

received or transmitted characters. The eight possible data formats are given in table 9.3. Note that these bits also enable the type of parity (if any) and the number of stop bits to be defined under software control, which is one of the nice features of a programmable peripheral. Possibly the most common data format for the transmission of information between a processor and a CRT terminal is: start bit + 7 data bits + even parity + 1 stop bit. The corresponding value of CR4, CR3, CR2 is 0, 1, 0.

The *transmitter control* field, CR5 and CR6, selects the state of the active-low request to send (RTS*) output and determines whether or not the transmitter section of the ACIA may generate an interrupt by asserting its IRQ* output. In most systems, RTS* is active-low whenever the ACIA is transmitting, because RTS* is used to activate equipment connected to the ACIA. The programming of the transmitter interrupt enable, and for that matter the receiver interrupt enable, is very much a function of the operating mode of the ACIA. If the ACIA is operated in a polled-data mode, interrupts are not necessary.

If the transmitter interrupt is enabled, an interrupt is generated by the transmitter whenever the transmit data register (TDR) is empty, signifying the need for new data from the CPU. When the ACIA's clear to send (CTS*) input is inactive-high, the TDR empty flag of the status register is held low, inhibiting any transmitter interrupt.

Setting both CR6 and CR5 to a logical 1 simultaneously creates a special case. When both these bits are high, a *break* is transmitted by the transmitter data output pin. A break is a condition in which the transmitter output is held at the active level (i.e., space or TTL logical zero) continuously. This state may be employed to force an interrupt at a distant receiver, because the asynchronous serial format precludes the existence of a space level for longer than about ten bit periods. The term *break* originates from the old current-loop data transmission system when a break was affected by disrupting (i.e., breaking) the flow of current round a loop.

The *receiver interrupt enable* field consists of 1 bit, CR7, which enables the generation of interrupts by the reviever when it is set (CR7 = 1) and disables receiver interrupts when it is clear (CR7 = 0). The receiver asserts its IRQ* output, assuming CR7 = 1, when the receiver data register full (RDRF) bit of the status register is set, indicating the presence of a new data character ready for the CPU to read. Two other circumstances also force a receiver interrupt. An overrun (see later) sets the RDRF bit and generates an interrupt. Finally, a receiver interrupt can also be generated by a low-to-high transition at the active-low data carrier detect (DCD*) input, signifying a loss of the carrier from a modem. Note that CR7 is a composite interrupt enable bit and enables all the three forms of receiver interrupt described previously. To enable either an interrupt caused by the RDR being full or an interrupt caused by a positive transition at the DCD* pin alone is impossible.

Status Register

The 8 bits of the read-only status register are depicted in table 9.4 and serve to indicate the status of both the transmitter and receiver portions of the ACIA at any instant.

TABLE 9.4 Format of the status register

BIT	SR7	SR6	SR5	SR4	SR3	SR2	SR1	SR0
Function	IRQ	PE	OVRN	FE	CTS	DCD	TDRE	RDRF

SR0—Receiver Data Register Full (RDRF) When set, the RDRF bit indicates that the receiver data register (RDR) is full and a new word has been received. If the receiver interrupt is enabled by CR7 = 1, a logical one in SR0 also sets the interrupt status bit SR7 (i.e., IRQ). The RDRF bit is cleared either by reading the data in the receiver data register or by carrying out a software reset on the control register. Whenever the data carrier detect (DCD*) input is inactive-high, the RDRF bit remains clamped at a logical zero, indicating the absence of any valid input.

SR1—Transmitter Data Register Empty (TDRE) The TDRE bit is the transmitter counterpart of the RDRF bit, SR0. A logical 1 in SR1 indicates that the contents of the transmit data register (TDR) have been sent to the transmitter and that the register is now ready to transmit new data. TDRE is cleared either by loading the transmit data register or by performing a software reset. If the transmitter interrupt is enabled, a logical one in bit SR1 (i.e., TDRE) also sets bit SR7 of the status word. Note again that SR7 is a composite interrupt bit because it is also set by an interrupt originating from the receiver side of the ACIA. If the clear to send (CTS*) input is inactive-high, the TDRE bit is held low, indicating that the terminal equipment is not ready for data.

SR2—Data Carrier Detect (DCD) This status bit, associated with the receiver side of the ACIA, is normally employed when the ACIA is connected to the telephone network via a modem. Whenever the DCD* input to the ACIA is inactive-high, SR2 is set. A logical one on the DCD* line generally signifies that the incoming serial data is faulty, which also has the effect of clearing the SR0 (i.e., RDRF) bit, as possible erroneous input should not be interpreted as valid data.

When the DCD* input makes a low-to-high transition, not only is SR2 set but the composite interrupt request bit, SR7, is also set if the receiver interrupt is enabled. Note that SR2 remains set even if the DCD* input later returns active-low. This action traps any occurrence of DCD* high, even if it goes high only briefly. To clear SR2, the CPU must read the contents of the status register and then the contents of the data register.

SR3—Clear to Send (CTS) The CTS* bit directly reflects the status of the CTS* input on the ACIA's transmitter side. An active-low level on the CTS* input indicates that the transmitting device (modem, paper tape punch, teletype, cassette recorder, etc.) is ready to receive serial data from the ACIA. If the CTS* input and therefore the CTS* status bit are high, the transmit data register empty bit, SR1, is inhibited (clamped at a logical zero) and no data may be transmitted by the ACIA. Unlike the DCD* status bit, the logical value of the CTS* status bit is determined only by the CTS* input and is not affected by any software operation on the ACIA.

SR4—Framing Error (FE) A framing error is detected by the absence of a stop bit and indicates a synchronization (i.e., timing) error, a faulty transmission, or a break condition. The framing error status bit, SR4, is set whenever the ACIA determines that a received character is incorrectly framed by a start bit and a stop bit. The framing error status bit is automatically cleared or set during the receiver data transfer time and is present throughout the time that the associated character is available. In other words, an FE bit is generated for each character received and a new character overwrites the old one's FE bit.

SR5—Receiver Overrun (OVRN) The receiver overrun status bit is set when a character is received by the ACIA but is not read by the CPU before a subsequent character is received, overwriting the last character, which is now lost. Consequently, the receiver overrun bit indicates that one or more characters in the data stream have been lost. The OVRN status bit is set at the midpoint of the last bit of the second character received in succession without a read of the RDR having occurred. Synchronization of the incoming data is not affected by an overrun error—the error is due to the CPU not having read a character, rather than by any fault in the transmission and reception process. The overrun bit is cleared after reading data from the RDR or by a software reset.

SR6—Parity Error (PE) The parity error status bit, SR6, is set whenever the received parity bit in the current character does not match the parity bit of the character generated locally in the ACIA from the received data bits. Odd or even parity may be selected by writing the appropriate code into bits CR2, CR3, and CR4 of the control register. If no parity is selected, then both the transmitter parity generator and receiver parity checker are disabled. Once a parity error has been detected and the parity error status bit set, it remains set as long as the erroneous data remains in the receiver register.

SR7—Interrupt Request (IRQ) The interrupt request status bit, SR7, is a composite active-high (note!) interrupt request flag, and is set whenever the ACIA wishes to interrupt the CPU, for whatever reason. The IRQ bit is set active-high by any of the following events:

1. Receiver data register full (SR0 set) and receiver interrupt enabled.
2. Transmitter data register empty (SR1 set) and transmitter interrupt enabled.
3. Data carrier detect status bit (SR2) set and receiver interrupt enabled.

Whenever SR7 is active-high, the active-low open-drain IRQ* output from the ACIA is pulled low. The IRQ bit is cleared by a read from the RDR, or by a write to the TDR, or by a software master reset.

Using the 6850 ACIA

The most daunting thing about many microprocessor interface chips is their sheer complexity. Often this complexity is more imaginary than real, because such peripherals are usually operated in only one of the many different modes that are software selectable. This fact is particularly true of the 6850 ACIA. Figure 9.8 shows how the 6850 is operated in a minimal mode. Only its serial data input (RxD) and output (TxD) are connected to an external system. The request to send (RTS*) output is left unconnected and clear to send (CTS*) and data carrier detect (DCD*) are both strapped to ground at the ACIA.

In a minimal (and noninterrupt) mode, bits 2 to 7 of the status register can be ignored. Of course, the error-detecting facilities of the ACIA are therefore thrown away. The software necessary to drive the ACIA in this minimal mode consists of three subroutines: an initialization, an input, and an output routine:

```
ACIAC       EQU      $E0001            Address of control/status register
ACIAD       EQU      ACIAC+2           Address of data register
RDRF        EQU      0                 Receiver data register full
TDRE        EQU      1                 Transmitter data register empty

INITIALIZE  MOVE.B   #%00000011,ACIAC  Reset the ACIA
            MOVE.B   #%00011001,ACIAC  Set up control word—disable
            RTS                        interrupts, RTS* low, 8 data
                                       bits, even parity, 1 stop bit,
                                       16 × clock

INPUT       BTST.B   #RDRF,ACIAC       Test receiver status
            BEQ      INPUT             Poll until receiver has data
            MOVE.B   ACIAD,D0          Put data in D0
            RTS

OUTPUT      BTST.B   #TDRE,ACIAC       Test transmitter status
            BEQ      OUTPUT            Poll until transmitter ready for data
            MOVE.B   D0,ACIAD          Transmit the data
            RTS
```

The INITIALIZE routine is called once before either input or output is carried out and has the effect of executing a software reset on the ACIA followed by setting up its control register. The control word %00011001 (see table 9.3) defines an 8-bit word with even parity and a clock rate (TxCLK, RxCLK) 16 times the data rate of the transmitted and received data.

The INPUT and OUTPUT routines are both entirely straightforward. Each tests the appropriate status bit and then reads data from or writes data to the ACIA's data register.

It is also possible to operate the ACIA in a minimal interrupt-driven mode. The IRQ* output is connected to one of the 68000's seven levels of interrupt request input and arrangements are made to supply the CPU with VPA* during an interrupt acknowledge cycle. Both transmitter and receiver interrupts are enabled by writing 1, 0, 1 into bits CR7, CR6, CR5 of the status register.

When a transmitter or receiver interrupt is initiated, it is still necessary to examine the RDRF and TDRE bits of the status register to determine that the ACIA did indeed request the interrupt and to separate transmitter and receiver requests for service. The effect of interrupt-driven I/O is to eliminate the time-wasting polling routines required by programmed I/O.

Figure 9.9 shows how the ACIA can be operated in a more sophisticated mode.

FIGURE 9.8 Minimal serial interface using the 6850 ACIA

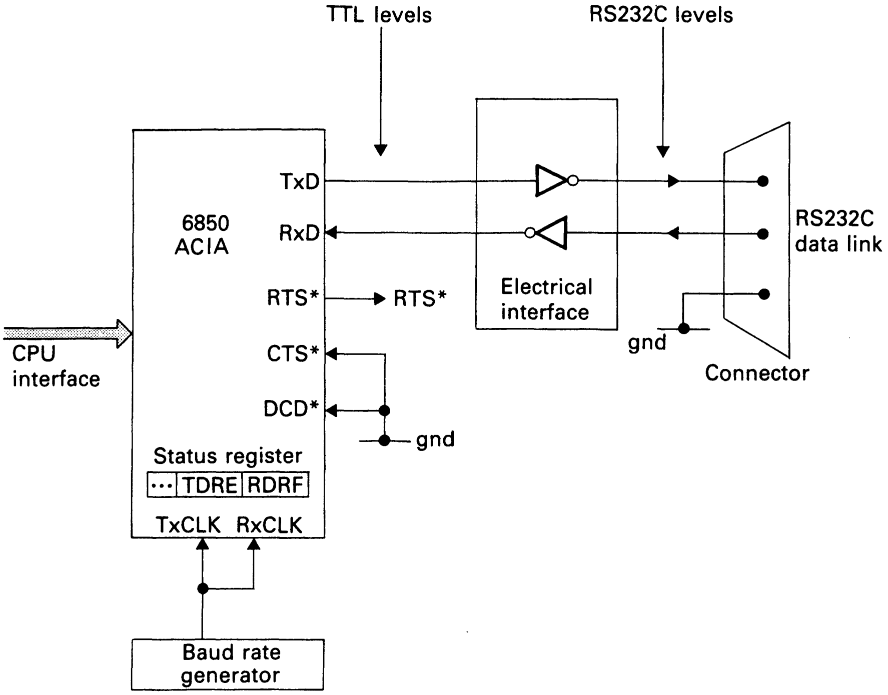
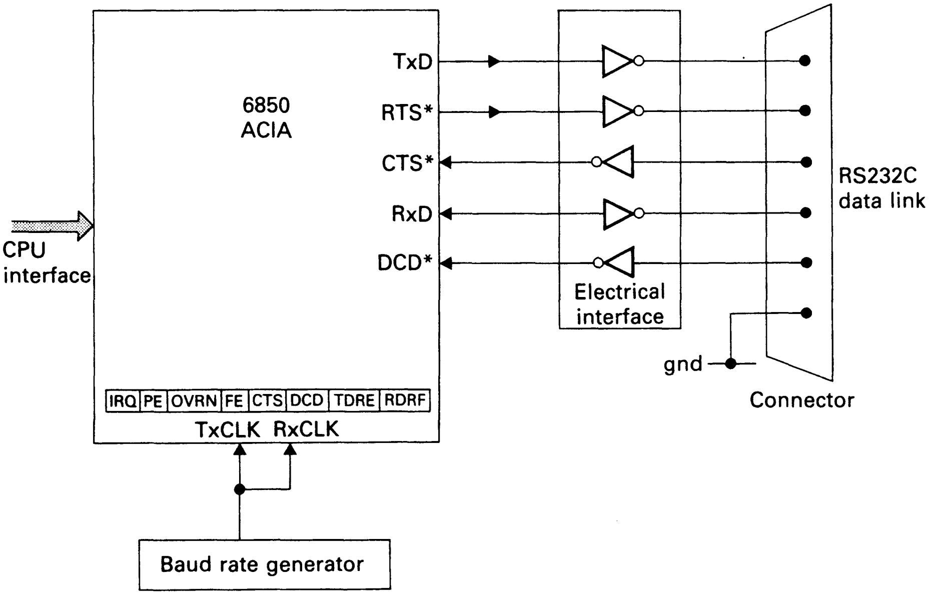


FIGURE 9.9 General-purpose serial interface using the 6850 ACIA



The reader may be tempted to ask, Why bother with a complex operating mode if the 6850 works quite happily in a basic mode? The answer is that the operating mode in figure 9.9 provides more facilities than the basic mode of figure 9.8.

In Figure 9.9 the transmitter side of the ACIA sends an RTS* signal and receives a CTS* signal from the remote terminal equipment. Now the remote equipment is able to say, "I am ready to receive your data," by asserting CTS*. In the cut-down mode of figure 9.8, the ACIA simply sends data and hopes for the best.

Similarly, the receiver side of the ACIA uses the data carrier detect (DCD*) input to signal the host computer that the receiver circuit us in a position to receive data. If DCD* is negated, the terminal equipment is unable to send data to the ACIA.

The software necessary to receive data when operating the 6850 in its more sophisticated mode is considerably more complex than that of the previous example. Provision for a full input routine is not possible here, as such a routine would include recovery procedures from the errors detected by the 6850 ACIA. These procedures are, of course, dependent on the nature of the system and the protocol used to move data between a transmitter and receiver. However, the following fragment of an input routine gives some idea of how the 6850's status register is used:

```
ACIAC        EQU      $<ACIA address>
ACIAD        EQU      ACIAC+2
RDRF         EQU      0              Receiver_data_register_full
TDRE         EQU      1              Transmitter_data_register_empty
DCD          EQU      2              Data_carrier_detect
CTS          EQU      3              Clear_to_send
FE           EQU      4              Framing_error
OVRN         EQU      5              Over_run
PE           EQU      6              Parity_error
*
INPUT        MOVE.B   ACIAC,D0       Get status from ACIA
             BTST     #RDRF,D0       Test for received character
             BNE.S    ERROR_CHECK    If character received, then test SR
             BTST     #DCD,D0        Else test for loss of signal
             BEQ      INPUT          Repeat loop while CTS clear
             BRA.S    DCD_ERROR      Else deal with loss of signal
ERROR_CHECK  BTST     #FE,D0         Test for framing error
             BNE.S    FE_ERROR       If framing error, deal with it
             BTST     #OVRN,D0       Test for overrun
             BNE.S    OVRN_ERROR     If overrun, deal with it
             BTST     #PE,D0         Test for parity error
             BNE.S    PE_ERROR       If parity error, deal with it
             MOVE.B   ACIAD,D0       Load the input into D0
             BRA.S    EXIT           Return
DCD_ERROR                            Deal with loss of signal
             BRA.S    EXIT
FE_ERROR                             Deal with framing error
             BRA.S    EXIT
OVRN_ERROR                           Deal with overrun error
             BRA.S    EXIT
PE_ERROR                             Deal with parity error
EXIT                  RTS
```

9.3 THE 68681 DUART

The 6850 ACIA is a first-generation interface device designed in the 1970s to work with the 8-bit 6800 microprocessor and is now rather outdated (although it is still widely used). Today's designers would rather implement an asynchronous serial interface with a more modern component, such as the 68681 DUART (dual universal asynchronous receiver/transmitter). The 68681 (from now on we will just call it "DUART") performs the same basic functions as a *pair* of 6850s plus a baud-rate generator. Designers prefer to use the DUART for the following reasons:

1. The DUART provides two independent asynchronous serial channels and therefore replaces two 6850 ACIAs.

2. The DUART has a full 68000 asynchronous bus interface, which means that it supports asynchronous data transfers and can supply a vector number during an interrupt acknowledge cycle.

3. The DUART has an on-chip programmable baud-rate generator, which saves both the cost and board space of a separate baud-rate generator. Moreover, the DUART's baud-rate generator can be programmed simply by loading an appropriate value into a clock select register. This feature makes it very easy to connect a system with a DUART to a communications system with an unknown baud rate. Communications systems based on the 6850 have to change their baud rate by altering links on the board, making it tedious to change the baud rate frequently. Note that the DUART can receive and transmit at different baud rates (as can the 6850).

4. The DUART has a quadruple buffered input which means that up to four characters can be received in a burst before the host processor has to read the input stream. The host computer has to read each character from a 6850 as it is received (otherwise an overrun will occur and characters will be lost). Similarly, the DUART has a double-buffered output, permitting one character to be transmitted while another is being loaded by the CPU.

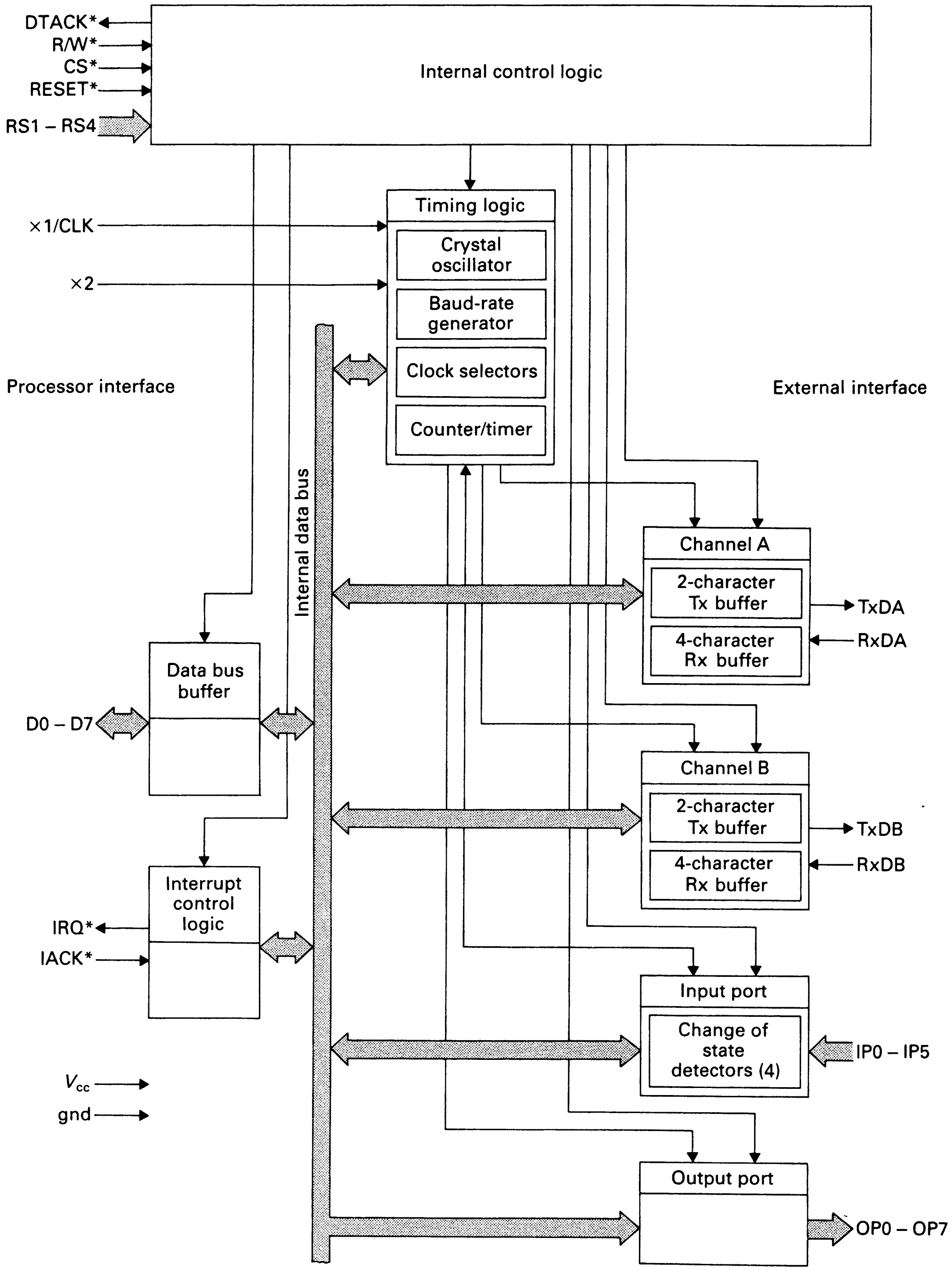
5. The DUART has 14 I/O pins (6 input, 8 output) that can be used as modem-control pins, clock input and outputs, or as general-purpose input/output pins.

6. The 6850 has just one operating mode. The DUART can support several modes (e.g., a self-test loopback mode).

Figure 9.10 illustrates the internal organization of a 68681 DUART.

Since the 68681 is so much better than the 6850, why, then, have we not used it to replace the 6850 in this chapter? The answer to this question is very simple. The 6850 is still widely used and is very much easier to understand than the more versatile DUART. However, since the DUART has become a standard in 68000-based systems, we cannot neglect it. The DUART is described in *Microprocessor Interfacing and the 68000* (see the bibliography), so the material is not repeated here.

FIGURE 9.10 The 68681 DUART



One way of approaching the DUART is to ignore its sophisticated functions and to treat it as an *advanced* ACIA. We will do this and describe how the DUART can be interfaced to a 68000 without going into fine detail. In short, we will treat the DUART as a black box.

The DUART's Registers

The DUART has 16 addressable registers, as illustrated in figure 9.11. Some registers are read-only, some are write-only, and some are read/write. For our current purposes, we concentrate on the DUART's five control registers that must be configured before it can be used as a transmitter or receiver (the 6850 has just a single control register). Note that some registers are *global* and affect the operation of both the DUART's serial channels, whereas others are *local* to channel A or to channel B (in what follows, we use channel A registers). These five control registers are: MR1A (master register 1), MR2A (master register 2), CSRA (clock select register), CRA (command register), and ACR (auxiliary control register). Note that MR1A and MR2A share the same address. After a reset, MR1A is selected at the base address of the DUART. When MR1A is loaded with data by the host processor, MR2A is automatically selected at the same address (you can access MR1A again only by resetting the DUART or by executing a special *select MR1A* command.

Figure 9.12 provides a simplified extract from the DUART's data sheet that describes the five control registers and the status register. Modes of no interest to us here, such as the DUART's parallel I/O capabilities, have not been included in figure 9.12. The following notes provide sufficient details about the DUART's registers to enable the reader to use it in its basic operating mode.

The *auxiliary control register*, ACR, selects the DUART's clock source (internal or external), selects its baud-rate set (there are two sets—setting ACR_7 to 0 selects set 1 and setting ACR_7 to 1 selects set 2), and controls certain parallel input pins. For our purposes, ACR can be loaded with $80 (to select baud-rate set 2) and ignored.

The *clock-select register(s)*, CSRA and CSRB, permit the programmer to select the DUART's baud rate (CSRA selects the channel A baud rate and CSRB the channel B baud rate). Figure 9.12 demonstrates that it is possible to select independent baud rates for transmission and reception. The values shown here can be loaded into the clock-select register to select the following popular baud rates (for both transmission and reception).

VALUE	BAUD RATE
44_{16}	300
55_{16}	600
66_{16}	1,200
88_{16}	2,400
99_{16}	4,800
BB_{16}	9,600
CC_{16}	19,200

FIGURE 9.11 The DUART's registers

RS4	RS3	RS2	RS1	Read (R/W* = 1)		Write (R/W* = 0)	
0	0	0	0	Mode register A	(MR1A, MR2A)	Mode register A	(MR1A, MR2A)
0	0	0	1	Status register A	(SRA)	Clock-select register A	(CSRA)
0	0	1	0	Do not access*		Command register A	(CRA)
0	0	1	1	Receiver buffer A	(RBA)	Transmitter buffer A	(TBA)
0	1	0	0	Input port change register	(IPCR)	Auxiliary control register	(ACR)
0	1	0	1	Interrupt status register	(ISR)	Interrupt mask register	(IMR)
0	1	1	0	Counter mode: current MSB of counter	(CUR)	Counter/timer upper register	(CTUR)
0	1	1	1	Counter mode: current LSB of counter	(CLR)	Counter/timer lower register	(CTLR)
1	0	0	0	Mode register B	(MR1B, MR2B)	Mode register B	(MR1B, MR2B)
1	0	0	1	Status register B	(SRB)	Clock-select register B	(CSRB)
1	0	1	0	Do not access*		Command register B	(CRB)
1	0	1	1	Receiver buffer B	(RBB)	Transmitter buffer B	(TBB)
1	1	0	0	Interrupt-vector register	(IVR)	Interrupt-vector register	(IVR)
1	1	0	1	Input port (unlatched)		Output port configuration register	(OPCR)
1	1	1	0	Start-counter command**		Output port Register (OPR)	Bit set command**
1	1	1	1	Stop-counter command**			Bit reset command**

* This address location is used for factory testing of the DUART and should not be read. Reading this location will result in undesired effects and possible incorrect transmission or reception of characters. Register contents may also be changed.

** Address triggered commands.

Each baud-rate value loaded into a clock-select register consists of two 4-bit values (bits 0–3 select the transmitter baud rate and bits 4–7 select the receiver baud rate). For example, the instruction MOVE.B #\$B8,CSRA selects a receive rate of 9,600 baud and a transmit rate of 2,400 baud.

The *channel mode control* registers define the operating mode of the DUART (MR1A, MR2A for channel A and MR1B, MR2B for channel B). Figure 9.12 provides a simplified account of these bits. To operate the DUART in its normal, 8-bit character mode with no parity, 1 stop bit, and no modem control functions activated, MR1A is loaded with 13_{16} and MR2A, with 07_{16}. Remember that these registers share the same address and that MR2A is selected automatically after MR1A has been loaded—that is,

```
MR1A    EQU    DUART_BASE
MR2A    EQU    MR1A                 MR1A, MR2A have same address
               MOVE.B #$13,MR1A     Load MR1A—no parity, 8 bits
               MOVE.B #$07,MR2A     Now load MR2A—normal mode, 1 stop bit
```

The *command registers* (CRA and CRB) permit the programmer to enable and disable a channel's receiver or transmitter and to issue certain commands to the DUART. The command CRA(6 : 4) = 001 resets the master register pointer to MR1A (since MR2A is automatically selected after MR1A has been loaded). You can load CRA with $0A_{16}$ to disable both channels during its setting up phase and then load it with 05_{16} to enable its transmitter and receiver ports once its other registers have been set up.

The DUART's *status registers* (SRA and SRB) are very similar to their 6850 counterpart. The major additions are SRA_7, which detects that a break has been received, SRA_3 (TxEMT), which indicates that the transmitter buffer is empty (i.e., there are no characters in the DUART's buffer waiting to be transmitted), and SRA_1 (FFULL), which indicates that the receiver buffer is full (there are four received characters waiting to be read). You can, of course, forget about these new bits and operate the DUART exactly like the ACIA just by using the TxRDY and RxRDY bits of its status register.

The difference between the status bits FFULL and TxRDY is that the FFULL flag is applied to the whole receiver buffer, whereas RxRDY tells us that there is at least one free place in the receiver buffer. Similarly, TxEMT tells us that there is no character in the transmitter buffer and that the buffer is completely empty, whereas TxRDY tells us that the DUART is ready for another character.

The DUART has sophisticated *interrupt control* and handling facilities (figure 9.13). The *interrupt vector register,* IVR, provides a vector number when the DUART generates an interrupt and receives an IACK response from the 68000. If the IVR has not been loaded by the programmer since the last time the DUART was reset, the DUART supplies an uninitialized vector number during an IACK cycle.

The DUART has two interrupt control registers with identical formats: ISR is an *interrupt status register* whose bits are set when "interrupt-generating" activities take place. IMR is an *interrupt mask register* whose bits are set by the programmer to enable an interrupt or cleared to mask the interrupt. For example, ISR_0 is set if

FIGURE 9.12 The DUART's control registers

Clock-select register A (CSRA)

Receiver-clock select			Transmitter-clock select		
Bit 7 Bit 6 Bit 5 Bit 4	Baud rate Set 1 ACR bit 7 = 0	 Set 2 ACR bit 7 = 1	Bit 3 Bit 2 Bit 1 Bit 0	Baud rate Set 1 ACR bit 7 = 0	 Set 2 ACR bit 7 = 1
0 0 0 0	50	75	0 0 0 0	50	75
0 0 0 1	110	110	0 0 0 1	110	110
0 0 1 0	134.5	134.5	0 0 1 0	134.5	134.5
0 0 1 1	200	150	0 0 1 1	200	150
0 1 0 0	300	300	0 1 0 0	300	300
0 1 0 1	600	600	0 1 0 1	600	600
0 1 1 0	1200	1200	0 1 1 0	1200	1200
0 1 1 1	1050	2000	0 1 1 1	1050	2000
1 0 0 0	2400	2400	1 0 0 0	2400	2400
1 0 0 1	4800	4800	1 0 0 1	4800	4800
1 0 1 0	7200	1800	1 0 1 0	7200	1800
1 0 1 1	9600	9600	1 0 1 1	9600	9600
1 1 0 0	38.4k	19.2k	1 1 0 0	38.4k	19.2k

Channel A mode register 1 (MR1A) and channel B mode register 1 (MR1B)

Rx RTS control	Rx IRQ* select	Error mode	Parity mode	Parity type	Bits-per-character
Bit 7	Bit 6	Bit 5	Bit 4 Bit 3	Bit 2	Bit 1 Bit 0
0 = Disabled 1 = Enabled	0 = RxRDY 1 = FFULL	0 = Char 1 = Block	0 0 = With parity 0 1 = Force parity 1 0 = No parity 1 1 = Multidrop mode*	With parity 0 = even 1 = odd Force parity 0 = Low 1 = High Multidrop mode 0 = Data 1 = Address	0 0 = 5 0 1 = 6 1 0 = 7 1 1 = 8

*The parity bit is used as the address/data bit in multidrop mode.

Channel A command register (CRA) and channel B command register (CRB)

Not used*	Miscellaneous commands	Transmitter commands	Receiver commands
Bit 7	Bit 6 Bit 5 Bit 4	Bit 3 Bit 2	Bit 1 Bit 0
X	0 0 0 No command 0 0 1 Reset MR pointer to MR1 0 1 0 Reset receiver 0 1 1 Reset transmitter 1 0 0 Reset error status 1 0 1 Reset channel's break-change interrupt 1 1 0 Start break 1 1 1 Stop break	0 0 No action, stays in present mode 0 1 Transmitter enabled 1 0 Transmitter disabled 1 1 Don't use, indeterminate	0 0 No action, stays in present mode 0 1 Receiver enabled 1 0 Receiver disabled 1 1 Don't use, indeterminate

*Bit seven is not used and may be set to either zero or one.

Channel A status register (SRA) and channel B status register (SRB)

Received break	Framing error	Parity error	Overrun error	TxEMT	TxRDY	FFULL	RxRDY
Bit 7*	Bit 6*	Bit 5*	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes	0 = No 1 = Yes

* These status bits are appended to the corresponding data character in the receive FIFO and are valid only when the RxRDY bit is set. A read of the status register provides these bits (seven through five) from the top of the FIFO together with bits four through zero. These bits are cleared by a reset error status command. In character mode, they are discarded when the corresponding data character is read from the FIFO.

Auxiliary control register (ACR)

BRG set Select*	Counter/timer mode and source**			Delta*** IP3 IRQ*	Delta*** IP2 IRQ*	Delta*** IP1 IRQ*	Delta*** IP0 IRQ*
Bit 7	Bit 6 Bit 5 Bit 4			Bit 3	Bit 2	Bit 1	Bit 0
0 = Set 1		Mode	Clock source	0 = Disabled	0 = Disabled	0 = Disabled	0 = Disabled
1 = Set 2	0 0 0	Counter	External (IP2)****	1 = Enabled	1 = Enabled	1 = Enabled	1 = Enabled
	0 0 1	Counter	TxCA – 1X clock of channel A transmitter				
	0 1 0	Counter	TxCB – 1X clock of channel B transmitter				
	0 1 1	Counter	Crystal or external clock (X1/CLK) divided by 16				
	1 0 0	Timer	External (IP2)****				
	1 0 1	Timer	External (IP2) divided by 16****				
	1 1 0	Timer	Crystal or external clock (X1/CLK)				
	1 1 1	Timer	Crystal or external clock (X1/CLK) divided by 16				

* Should only be changed after both channels have been reset and are disabled.

** Should only be altered while the counter/timer is not in use (i.e., stopped if in counter mode, output and/or interrupt masked if in timer mode).

*** Delta is equivalent to change-of-state.

**** In these modes, because IP2 is used for the counter/timer clock input, it is not available for use as the channel B receiver-clock input.

Channel A mode register 2 (MR2A) and channel B mode register 2 (MR2B)

Channel mode	Tx RTS control	CTS enable transmitter	Stop bit length			
Bit 7 Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
					6–8 bits/ character	5 bits/ character
0 0 = Normal	0 = disabled	0 = disabled	(0)	0 0 0 0 =	0.563	1.063
0 1 = Automatic echo	1 = enabled	1 = enabled	(1)	0 0 0 1 =	0.625	1.125
1 0 = Local loopback			(2)	0 0 1 0 =	0.688	1.188
1 1 = Remote loopback			(3)	0 0 1 1 =	0.750	1.250
			(4)	0 1 0 0 =	0.813	1.313
			(5)	0 1 0 1 =	0.875	1.375
			(6)	0 1 1 0 =	0.938	1.438
			(7)	0 1 1 1 =	1.000	1.500
			(8)	1 0 0 0 =	1.563	1.563
			(9)	1 0 0 1 =	1.625	1.625
			(A)	1 0 1 0 =	1.688	1.688
			(B)	1 0 1 1 =	1.750	1.750
			(C)	1 1 0 0 =	1.813	1.813
			(D)	1 1 0 1 =	1.875	1.875
			(E)	1 1 1 0 =	1.938	1.938
			(F)	1 1 1 1 =	2.000	2.000

NOTE:
If an external 1X clock is used for the transmitter, MR2 bit 3 = 0 selects one stop bit and MR2 bit 3 = 1 selects two stop bits to be transmitted.

FIGURE 9.13 The DUART's interrupt control registers

Interrupt vector register IVR

Interrupt vector bits D7 – D0

Interrupt status register ISR

7	6	5	4	3	2	1	0
Input port change	Delta break B	RxRDYB/ FFULLB	TxRDYB	Counter/ timer	Delta break A	RxRDYA/ FFULLA	TxRDYA

Interrupt mark register IMR

7	6	5	4	3	2	1	0
Input port change	Delta break B	RxRDYB/ FFULLB	TxRDYB	Counter/ timer	Delta break A	RxRDYA/ FFULLA	TxRDYA

NOTE:

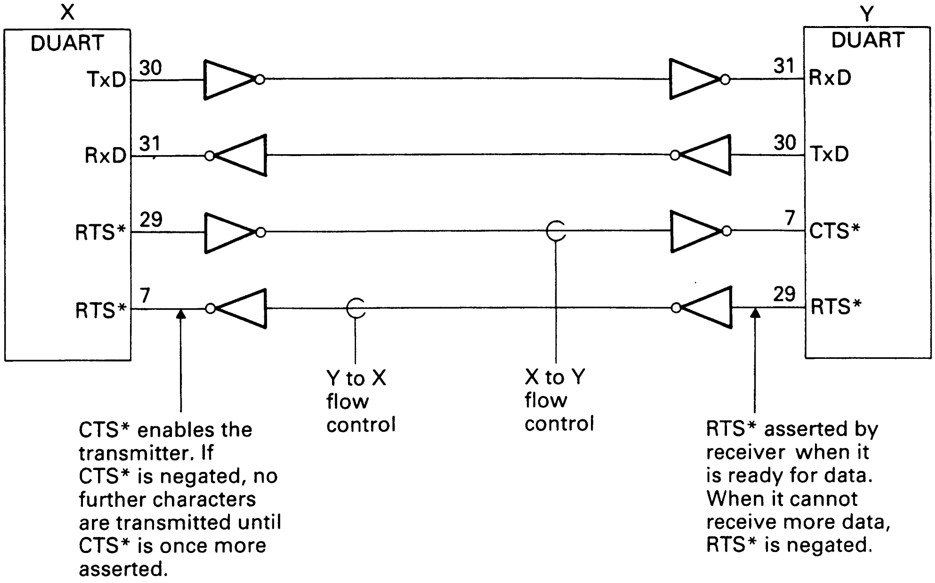
RxRDY/FFULL	Interrupt if RxRDY bit in status register set
TxRDY	Interrupt if TxRDY bit in status register set

Bit 6 of the channel mode register ($MR1A_6$ or $MR1B_6$) determines whether interrupt status register bits ISR_1 and ISR_5 are set on RxRDY or FFULL. If $MR1A_6 = 0$, ISR_1 is set on RxRDY (i.e., at least one character received). If $MR1A_6 = 1$, ISR_1 is set on FFULL (i.e., receiver buffer full).

TxRDYA is asserted to indicate that the channel A transmitter is ready for a character. If IMR_0 is set to 1, the DUART will generate an interrupt when channel A is ready to transmit a character.

We said earlier that the DUART has multipurpose I/O pins, which can be used as simple I/O pins or to perform special functions. Input pins IP0–IP5 are configured by bits in the CSRA/B and ACR registers. These pins can be programmed to provide inputs for the DUART's timer/counter, its baud-rate generator, and its clear to send modem control. When $MR2A_4 = 1$, pin IP0 acts as a channel A active-low clear to send input. Similarly, IP1 can be configured as channel B's CTS* input by setting $MR2B_4$ to 1. If the DUART is programmed to use its CTS* pins (i.e., $MRSA/B_4 = 1$), data is not transmitted by the DUART whenever CTS* is high—that is, the remote receiver can negate CTS* to stop the DUART sending further data. Figure 9.14 demonstrates how CTS* is used in conjunction with RTS* (see the following discussion).

FIGURE 9.14 Flow control and the DUART


X
DUART
TxD
RxD
RTS*
RTS*
30
31
29
7
Y
DUART
RxD
TxD
CTS*
RTS*
31
30
7
29
Y to X flow control
X to Y flow control
CTS* enables the transmitter. If CTS* is negated, no further characters are transmitted until CTS* is once more asserted.
RTS* asserted by receiver when it is ready for data. When it cannot receive more data, RTS* is negated.

NOTE the following steps to configure the DUART to perform flow control:

```
*  Set MR2A$_4$ to 1 and MR2A$_5$ to 1 to configure OP0 as
*  RTS output
*
   MOVE.B      #$83,MR1A
   MOVE.B      #$27,MR2A
*
*
*  Note that RTS* must initially be asserted
*     manually—after that, RTS* is asserted
*     automatically whenever the receiver is ready to
*     receive more data. Note also that the contents
*     of the DUART's output port register are
*     inverted before they are fed to the output
*     pins. That is, to assert RTS* low, it is
*     necessary to load a one into the appropriate
*     bit of the OPR.
*
   MOVE.B      #$01,OPR      Set OPR0 to assert RTS*
```

The 8-bit output port is controlled by an output port configuration register (OPCR) and certain bits of the ACR, MR1A, MR2A, MR1B, and MR2B registers. Output bits can be programmed as simple outputs cleared and set under programmer control, timer and clock outputs, and status outputs. Some of the output functions that can be selected are the following:

PIN	FUNCTION	ACTION
OP0	RxRTSA*	Asserted if channel A Rx is able to receive a character
OP0	TxRTSA*	Negated if channel A Tx has nothing to transmit
OP1	RxRTSB*	Asserted if channel B Rx is able to receive a character
OP1	TxRTSB*	Negated if channel B Tx has nothing to transmit
OP4	RxRDYA	Asserted if channel A Rx has received a character
OP5	RxRDYB	Asserted if channel B Rx has receiveda character
OP6	TxRDYA	Asserted if channel A Tx ready for data
OP7	TxRDYB	Asserted if channel B Tx ready for data

Note the difference between the RxRTS* and TxRTS* functions. RxRTS* is used by a receiver to indicate to the remote transmitter that it (the receiver) is able to accept data. RxRTS* is connected to the transmitter's CTS* input to perform flow control (figure 9.14). The TxRTS* function is used to indicate to a modem that the DUART has further data to transmit.

Programming the 68681 DUART

Once the DUART has been configured, it can be used to transmit and receive characters exactly like the 6850. The following fragment of code provides basic initialization, receive, and transmit routines for the DUART.

```
*          DUART equates
MR1A       EQU     1                     Mode register 1
MR2A       EQU     1                     Mode register 2 (same address as MR1A)
SRA        EQU     3                     Status register
CSRA       EQU     3                     Clock select register
CRA        EQU     5                     Command register
RBA        EQU     7                     Receiver buffer register (i.e., serial in)
TBA        EQU     7                     Transmitter buffer register (i.e., data out)
IPCR       EQU     9                     Input port change register
ACR        EQU     9                     Auxiliary control register
ISR        EQU     11                    Interrupt status register
IMR        EQU     11                    Interrupt mask register
IVR        EQU     25                    Interrupt vector register
*
*          Initialize the DUART
*
INITIAL    LEA     DUART,A0             A0 points at DUART base address
*
*          Note the following three instructions are not necessary
*          after a hardware reset to the DUART. They are included to
*          show how the DUART is reset.
*
           MOVE.B  #$30,CRA(A0)         Reset port A transmitter
           MOVE.B  #$20,CRA(A0)         Reset port A receiver
           MOVE.B  #$10,CRA(A0)         Reset port A MR (mode register) pointer
*
*          Select baud rate, data format and operating modes by
*          setting up the ACR, MR1 and MR2 registers
```

```
*
            MOVE.B      #$00,ACR(A0)       Select baud-rate set 1
            MOVE.B      #$BB,CSRA(A0)      Set both Rx and Tx speeds to 9600 baud
            MOVE.B      #$93,MR1A(A0)      Set port A to 8 bits, no parity, 1 stop bit,
*                                          enable RxRTS output
            MOVE.B      #$37,MR2A(A0)      Select normal operating mode, enable
*                                          TxRTS, TxCTS, 1 stop bit
            MOVE.B      #$05,CRA(A0)       Enable port A transmitter and receiver
            RTS
*
*           Input a single character from port A (polled mode) into D2
*
PUT_CHAR    MOVEM.L     D0-D1/A0,-(SP)     Save working registers
            LEA         DUART,A0           A0 points to DUART base address
Input_poll  MOVE.B      SRA(A0),D1         Read the port A status register
            BTST        #RxRDY,D1          Test receiver ready status
            BEQ         Input_poll         UNTIL character received
            MOVE.B      RBA(A0),D2         Read the character received by port A
            MOVEM.L     (SP)+,D0-D1/A0     Restore working registers
            RTS
*
*           Transmit a single character in D0 from port A (polled mode)
*
PUT_CHAR    MOVEM.L     D0-D1/A0,-(SP)     Save working registers
            LEA         DUART,A0           A0 points to DUART base address
Output_poll MOVE.B      SRA(A0),D1         Read port A status register
            BTST        #TxRDY,D1          Test transmitter ready status
            BEQ         Output_poll        UNTIL transmitter ready
            MOVE.B      D0,TBA(A0)         Transmit the character from port A
            MOVEM.L     (SP)+,D0-D1/A0     Restore working registers
            RTS
*
```

In spite of the DUART's complexity, you can see that it may be operated in a simple, noninterrupt-driven, character-by-character input/output mode, exactly like the ACIA, once its registers have been set up. On at least one occasion we have tested software written for a 6850-based system on a board with a DUART by making the following modifications to the 6850's I/O routines.

```
                  6850 I/O                         DUART I/O
--------------------------------------------------------------------------
SETUP    LEA        ACIA,A0          SETUP    LEA        DUART,A0
         MOVE.B     #$03,(A0)                 MOVE.B     #$13,(A0)
         MOVE.B     #$15,(A0)                 MOVE.B     #$07,(A0)
         RTS                                  MOVE.B     #$BB,2(A0)
                                              MOVE.B     #$05,4(A0)
                                              RTS

         LEA        ACIA,A0                   LEA        DUART,A0
INPUT    BTST.B     #0,0(A0)         INPUT    BTST.B     #0,2(A0)
         BNE        INPUT                     BNE        INPUT
         MOVE.B     2(A0),D0                  MOVE.B     6(A0),D0
         RTS                                  RTS

OUTPUT   BTST.B     #1,0(A0)         OUTPUT   BTST.B     #2,2(A0)
         BNE        OUTPUT                    BNE        OUTPUT
         MOVE.B     D0,2(A0)                  MOVE.B     D0,6(A0)
```

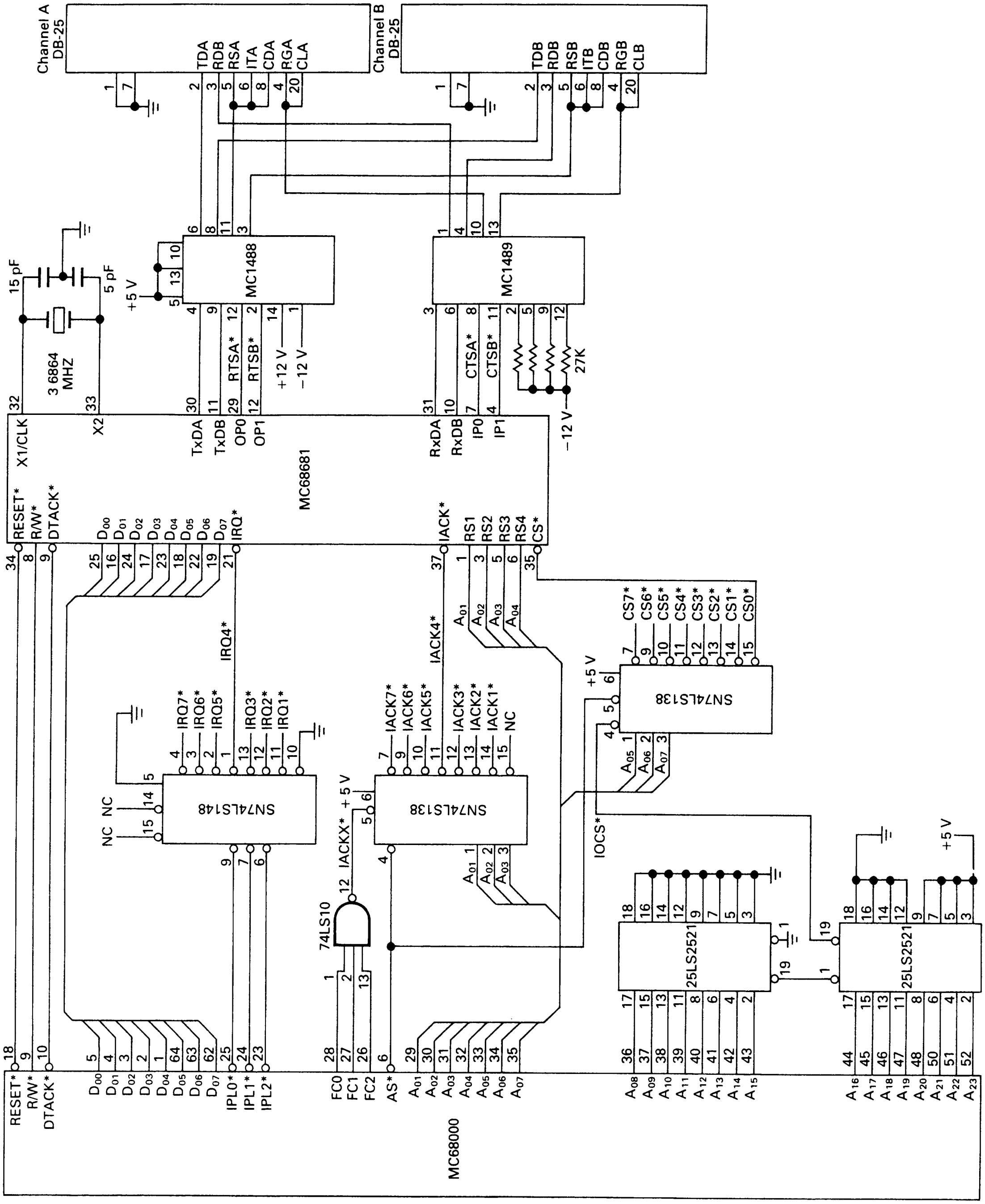


FIGURE 9.15 Interface between a DUART, a 68000 CPU, and two serial channels

The interface between a DUART and both a 68000 and two serial data links is illustrated in figure 9.15. As you can see, the DUART's 68000 interface is entirely conventional and requires no further comment. In this case, the DUART uses its internal baud-rate generator to supply clocks to both serial channels. The baud-rate generator uses a 3.6864-MHz quartz crystal, which is widely available.

The serial data links each provide a request to send output and a clear to send input to provide flow control of both transmitted and received data.

9.4 SYNCHRONOUS SERIAL DATA TRANSMISSION

The type of asynchronous serial data link described in section 9.1 is widely employed to link relatively slow peripherals such as printers and VDTs with processors. Where information has to be transferred between the individual computers of a network, synchronous serial data transmission is a more popular choice. In a synchronous serial data transmission system, the information is transmitted continuously without gaps between adjacent groups of bits. We use the expression *groups of bits* because synchronous systems can transmit entire blocks of pure binary information at a time, rather than transmitting information as a sequence of ASCII-encoded characters. Before continuing, we need to point out that synchronous serial data links are often used in a much more sophisticated way than their asynchronous counterparts, which simply move data between a processor and its peripheral. This section covers only the basic details of a synchronous serial data link.

Two problems face the designer of a synchronous serial system. One is how to divide the incoming data stream into individual bits and the other is how to divide the data bits into meaningful units.

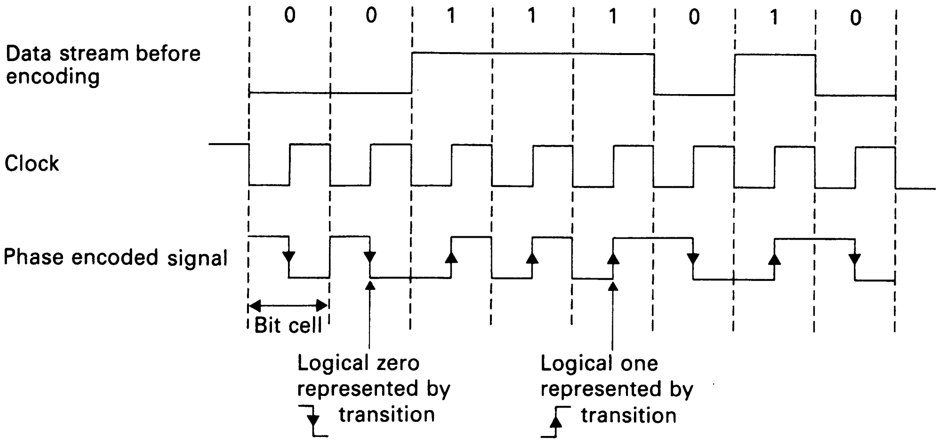
Bit Synchronization

As synchronous serial data transmission involves very long (effectively infinite) streams of data elements, the clocks at the transmitting and receiving ends of a data link must therefore be permanently synchronized. If a copy of the transmitter's clock were available at the receiver, no difficulty would be encountered in breaking up the data stream into individual bits. As this arrangement requires an extra transmission path between the transmitter and the receiver, it is not a popular solution to the problem of bit synchronization.

A better solution is found by encoding the data to be transmitted in such a way that a synchronizing signal is included with the data signal. Here, we do not delve deeply into the ways in which this situation may be achieved but show one popular arrangement.

The basic method of extracting a timing signal from synchronous serial data is illustrated in figure 9.16. The serial data stream is combined with a clock signal to give the encoded signal, which is actually transmitted over the data link. The encod-

FIGURE 9.16 Phase-encoded synchronous serial bit stream


0
0
1
1
1
0
1
0
Data stream before encoding
Clock
Phase encoded signal
Bit cell
Logical zero represented by transition
Logical one represented by transition

ing algorithm is simple. A logical one is represented by a positive transition in the center of a bit cell and a logical zero by a negative transition. This form of encoding is called phase encoding (PE) or Manchester encoding and is widely used. At the receiver, the incoming data can readily be split into a clock signal and a data signal. Integrated circuits that modulate or demodulate Manchester-encoded signals are readily available.

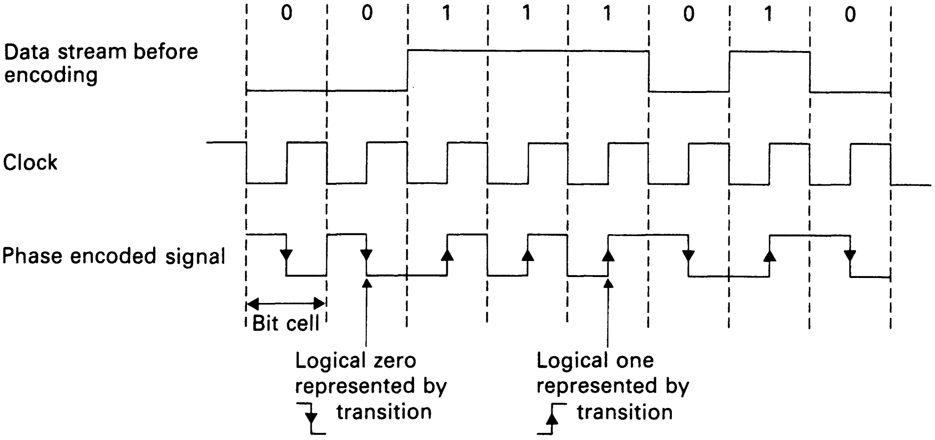
Word Synchronization

Having divided the incoming stream into individual data elements (i.e., bits), the next step is to group the bits together into meaningful units. We have called these *words*, although they may vary from 8 bits long to thousands of bits long. At first sight, dividing a continuous stream of bits into individual groups of bits might appear to be a most difficult task. Infactitisquiteaneasytasktoformbitsintowords. Here we have deleted interword spacing in the plain text, making it harder, but not impossible, to read. The reader examines the string of letters and looks for recognizable groups corresponding to valid words in English. A similar technique can be applied to continuous streams of binary data. Two basic modes of operation of synchronous serial data links exist: character oriented and bit oriented. In the former, the data stream is divided into separate characters and in the latter it is divided into much longer blocks of pure binary data.

Character-Oriented Data Transmission

In character-oriented data transmission systems, the information to be transmitted is encoded in the form of (usually) ASCII characters. One of the most popular character-oriented systems is called BISYNC, or binary synchronous data transmis-

FIGURE 9.16 Phase-encoded synchronous serial bit stream


0
0
1
1
1
0
1
0
Data stream before encoding
Clock
Phase encoded signal
Bit cell
Logical zero represented by transition
Logical one represented by transition

ing algorithm is simple. A logical one is represented by a positive transition in the center of a bit cell and a logical zero by a negative transition. This form of encoding is called phase encoding (PE) or Manchester encoding and is widely used. At the receiver, the incoming data can readily be split into a clock signal and a data signal. Integrated circuits that modulate or demodulate Manchester-encoded signals are readily available.

Word Synchronization

Having divided the incoming stream into individual data elements (i.e., bits), the next step is to group the bits together into meaningful units. We have called these *words*, although they may vary from 8 bits long to thousands of bits long. At first sight, dividing a continuous stream of bits into individual groups of bits might appear to be a most difficult task. Infactitisquiteaneasytasktoformbitsintowords. Here we have deleted interword spacing in the plain text, making it harder, but not impossible, to read. The reader examines the string of letters and looks for recognizable groups corresponding to valid words in English. A similar technique can be applied to continuous streams of binary data. Two basic modes of operation of synchronous serial data links exist: character oriented and bit oriented. In the former, the data stream is divided into separate characters and in the latter it is divided into much longer blocks of pure binary data.

Character-Oriented Data Transmission

In character-oriented data transmission systems, the information to be transmitted is encoded in the form of (usually) ASCII characters. One of the most popular character-oriented systems is called BISYNC, or binary synchronous data transmission. Take, for example, the four-character string "Alan;" it would be sent as the sequence of four 7-bit characters. The individual letters are ASCII encoded as:

A = $41

l = $6C

a = $61

n = $6E

Putting these together and reading the data stream from left to right with the first bit representing the least significant bit of the "A," we get:

1000001001101110000110111011

Some method is needed of identifying the beginning of a message. Once this has been done, the bits can be divided into characters by arranging them into groups of seven (or eight if a parity bit is used) for the duration of the message.

The ASCII code includes a number of characters specifically designed to control the flow of data over a synchronous serial data link. One such character is SYN (as in SYNchronization), whose code is $16, or 0010110. SYN is used to denote the beginning of a message. The receiver reads the incoming bits and looks for the string 0010110, representing a SYN and therefore the start of a message. Unfortunately, such a simple scheme is fatally flawed. The end of one character might be combined with the neginning of the following character to create a false SYN pattern. To avoid this situation, two SYN characters are transmitted sequentially. The receiver reads the first SYN and then looks for the second. If the receiver does not find another SYN, it assumes a false synchronization and continues looking for a valid SYN.

In addition to the synchronization character, the ASCII code provides other characters, such as STX (start of text), to help the user format data into meaningful units. However, character-oriented data transmission systems are not as popular as bit-oriented systems and are therefore not dealt with further here.

Bit-Oriented Data Transmission

Although the ASCII code is excellent for representing text, it is ill-fitted to the representation of pure binary data. Pure binary data can be anything from a core dump (a block of memory) or a program in binary form to floating-point numbers. When data is represented in character form, choosing one particular character (e.g., SYN) as a special marker is easy. When the data is in a pure binary form, choosing any particular data word as a reserved marker or flag is apparently impossible. Bit-oriented protocols (BOPs) have been devised to handle pure binary data.

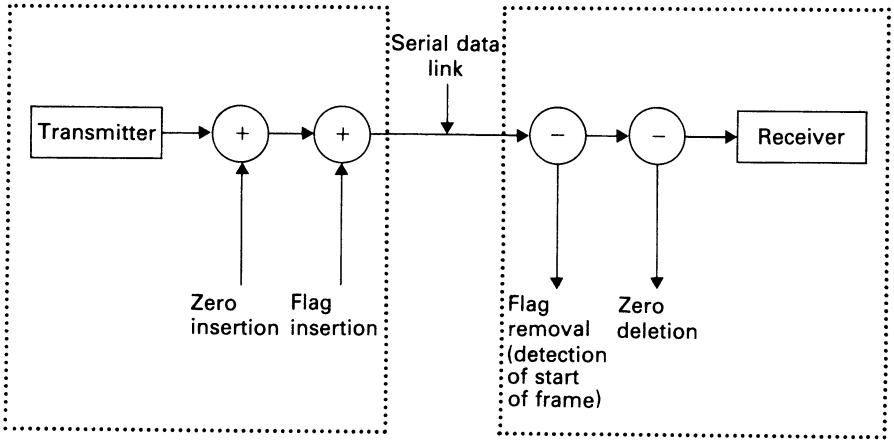
Fortunately, a remarkably simple and very elegant technique can be used to solve this problem. The beginning of each new block of data, called a frame, is denoted by the special (i.e., unique) binary sequence 01111110. Whenever the receiver detects this pattern, it knows that it has found the start (or end) of a block of data. The special sequence 01111110 is called an *opening* or *closing flag*. Of course, we still have the problem of what to do if we wish to send the pattern 01111110 as part of the data stream to be transmitted. Clearly, it cannot be sent in the form it occurs naturally, as the receiver would regard it as an opening or closing flag.

The transmitter avoids the preceding problem by a process called *bit-stuffing*. Whenever the pattern 011111 is detected at the transmitter (i.e., five 1s in series), the transmitter says, "If the next two bits are a 1 followed by a 0, a spurious flag will be created." Therefore, the transmitter inserts (i.e., stuffs) a 0 after the fifth logical one in succession in order to avoid the generation of a flag pattern. In this way, a flag can never appear by accident in the transmitted data stream.

At the receiver, the incoming bit stream is examined and opening or closing flags are deleted from the data stream. If the sequence 0111110 is found, the 0 following the fifth logical 1 is deleted, as it *must* have been inserted at the transmitter. In this way, any bit pattern may be presented to the transmitter, as bit-stuffing prevents the accidental occurrence of the opening or closing flag. Figure 9.17 illustrates the process of bit-stuffing.

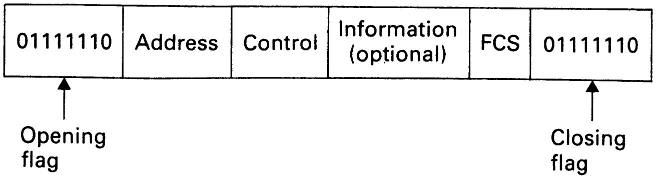
Modern bit-oriented synchronous serial data transmission systems have largely been standardized and use the HDLC data format. HDLC stands for *high-level data link control*. Information is transmitted in the form of packets or frames, with each packet separated by one or two flags as described previously. The format of a typical HDLC frame is given in figure 9.18. Following the opening flag is an address field of 8 bits, which defines the address of the secondary station (or slave) in situations

FIGURE 9.17 Process of bit-stuffing



Example:	0111110	→	01111100	→	0111110
	0111111	→	01111101	→	0111111
	01111111	→	011111011	→	01111111
	Original data		Data with stuffed zero		Data after zero deletion

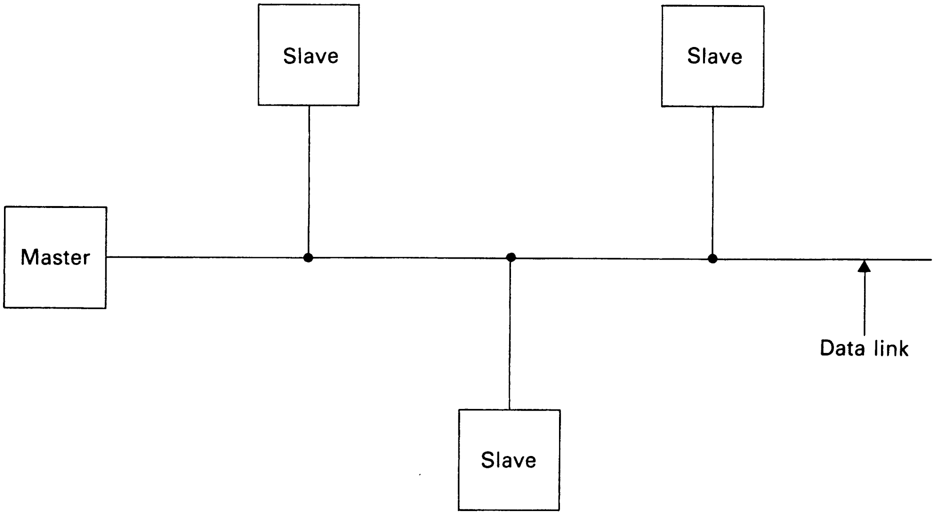
FIGURE 9.18 High-level data link control format



where a master station may be in communication with several slaves. An address field allows the master to send a message to one of its slaves without ambiguity. Any slave receiving a message whose address does not match that in the address field of a frame ignores that frame. Figure 9.19 shows the arrangement of a typical master-slave system. Remember that we have already stated that synchronous serial transmission systems are frequently used in more sophisticated ways than their asynchronous counterparts.

Following the address field is an 8-bit control field that controls the operation of the data link. The purpose of this field is to permit an orderly exchange of messages and to help detect and deal with lost messages. All that need be said here is that the control field provides the HDLC scheme with some very powerful facilities that are almost entirely absent in simple synchronous serial data links described earlier. The control field is followed by an optional data field (information field, or I field) containing the data to be transmitted. The I field is optional because frames may be transmitted for purely control purposes without an I field. Immediately after the I field (or control field if the I field is absent) comes the frame check sequence, FCS, which is a very powerful error-detecting code of 16 bits' length that is able to

FIGURE 9.19 Master-slave data transmission



detect the vast majority of errors (single or multiple bit) in the preceding fields. We are not able to go into detail about the theory of the FCS here, but the following notes should help.

The p bits in the packet, or frame, between the opening flag and the FCS itself are regarded as forming the coefficients of a polynomial of degree p. This polynomial is divided by a standard polynomial using modulo two arithmetic to yield a quotient and a 16-bit remainder. The quotient is discarded and the remainder forms the 16-bit FCS. At the receiver, the bits between the opening flag and received FCS are divided by the same generator polynomial to yield a local FCS. If the local FCS is the same as the received FCS, we can assume that the frame is free from all transmission errors. If they differ, the current frame is rejected.

Following the FCS is a closing flag, 01111110. Some arrangements require a closing flag for the current frame to be followed by an opening flag for the next frame. Other systems use one flag both to close the current frame and to open the next frame.

Clearly, a synchronous system is more efficient than an asynchronous system because of the absence of start, parity, and stop bits for each transmitted character. However, the real advantage of a synchronous system combined with the HDLC frame structure is its ability to control a data transmission system.

9.5 SERIAL INTERFACE STANDARDS (RS-232 AND RS-422/RS-423)

Because of the low cost of a serial interface and transmission path, the serial data link is used to connect a very wide range of peripherals to computer equipment. Once only teletypes and modems were likely to be connected to a computer by serial data links. Today, almost any peripheral, from CRT terminals to graphics tablets to disk drives, may use serial data links. Therefore, the data link should be standardized so that a peripheral from one manufacturer can be plugged into the serial port of a computer from another manufacturer.

Such a serial interface standard has been created by the Electronic Industries Association (EIA) and is known as the RS-232C serial interface.

The RS-232C Serial Interface

The EIA RS-232C standard was largely intended to link data terminal equipment (DTE) with data communications equipment (DCE). The DCE corresponds to the computer or terminal and the DTE corresponds to the modem or similar line equipment. RS-232C specifies the electrical and mechanical aspects of the serial interface together with the functions of the signals forming the interface. In theory, any RS-232C-compatible DCE can be connected directly to any RS-232C-compatible DTE.

Alas, the life of the computer technician is filled with time-wasting requests by programmers to get their printer to work with their computer—both of which have "RS-232C" serial interfaces. We put "RS-232C" in quotation marks because many (the majority?) equipment suppliers implement their own subset of an RS-232C interface. The likelihood that the subset of the RS-232C interface in the printer is incompatible with the subset in the computer is highly probable.

Mechanical Interface

Fortunately, the vast majority of equipment suppliers adhere to the mechanical aspects of the RS-232C standard and use the D-type connectors illustrated in figure 9.20. This connector is available in 9-, 15-, 25-, 37-, and 50-pin versions, but only the 25-way D connector may be used with RS-232C standard serial data links. The pinout of this connector is given in table 9.5, although we must appreciate the fact that very few implementations of the RS-232C standard implement the full standard.

FIGURE 9.20 D-type connector

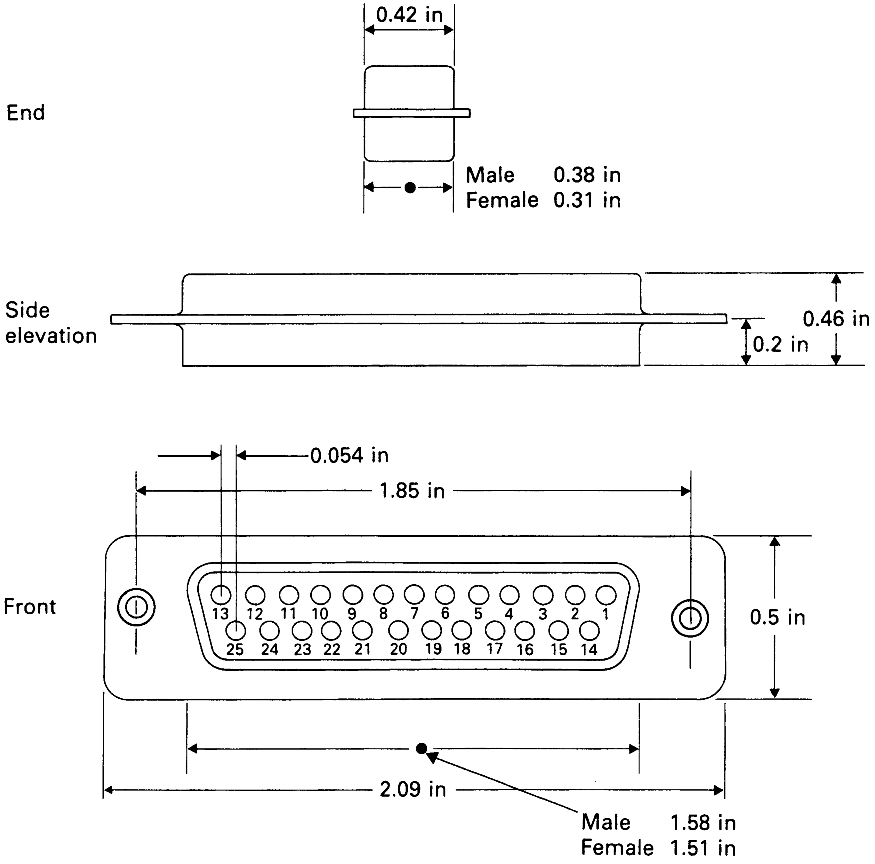


TABLE 9.5 Pinout of the RS-232C 25-way connector

PIN	NAME	FUNCTION
1	Protective ground	Electrical equipment frame and dc power ground
2	Transmitted data	Serial data generated by the DTE
3	Received data	Serial data generated by the DCE
4	Request to send	When asserted indicates that the DTE is ready to transmit primary data
5	Clear to send	When asserted indicates that the DCE is ready to transmit primary data
6	Data set ready	When asserted indicates that the DCE is not in a test, voice, or dial mode, that all initial handshake, answer tone, and timing delays have expired
7	Signal ground	Common ground reference for all circuits except protective ground
8	Received line signal detector	When asserted indicates that carrier signals are being received from the remote equipment
9	Reserved	
10	Reserved	
11	Unassigned	
12	Secondary received line signal detector	When asserted indicates that the secondary channel data carrier signals are being received from the remote equipment
13	Secondary clear to send	When asserted indicates that the DCE is ready to transmit secondary data
14	Secondary transmitted data	Low-speed secondary data channel generated by the DTE
15	Transmitted signal element timing	The signal on this line provides the DTE with signal element timing information
16	Secondary received data	Low-speed secondary channel data generated by the DCE
17	Receiver signal element timing	The signal on this line provides the DTE with signal element timing information
18	Unassigned	
19	Secondary request to send	When asserted indicates that the DTE is ready to transmit secondary channel data
20	Data terminal ready	When asserted indicates that the data terminal is ready
21	Signal quality detector	When asserted indicates that the received signal is probably error free; when negated indicates that the received signal is probably in error
22	Ring indicator	When asserted indicates that modem has detected a ringing tone on the telephone line
23	Data signal rate detector	Selects between two possible data rates
24	Transmit signal element timing	The signal on this line provides the DCE with signal element timing information
25	Unassigned	

Electrical Interface

The RS-232C standard is intended to provide serial communication facilities over relatively short distances and its electrical specifications reflect this. Table 9.6 gives the basic electrical parameters of the standard and figure 9.21 shows how the electrical interface may be implemented.

The circuit of figure 9.21 uses a single-ended bipolar unterminated circuit; that is, the circuit is single-ended (i.e., unbalanced) because the signal level to be transmitted is referred to ground and one of the signal-carrying conductors is grounded at both ends of the data link. The circuit is unterminated because no requirement exists in the RS-232C standard to match the characteristic impedance of the receiver to that of the transmission path.

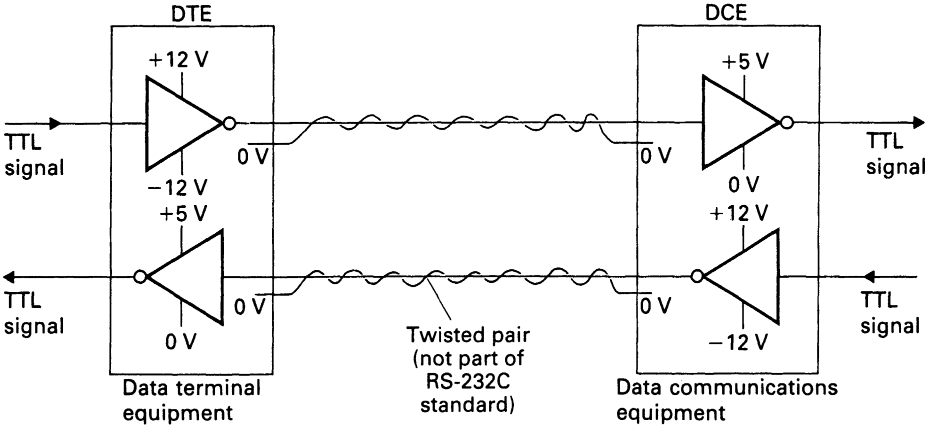
One of the key parameters in table 9.6 is the receiver maximum input threshold of -3 to $+3$ V. A space is guaranteed to be recognized if the input is more positive than $+3$ V and a mark is guaranteed to be recognized if the input is more negative than -3 V. The threshold separating mark and space levels is truly massive. Unless a transmitter can produce a voltage swing at the end of a transmission path of greater than 6 V, the received signal falls outside the minimum requirements of RS-232C. However, most real receivers for RS-232C signals have *practical* input thresholds well below -3 to $+3$ V. Therefore, as most engineers have noticed, it is often possible to have much longer transmission paths than the standard stipulates.

Interfacing to RS-232C lines is now very easy, as the major semiconductor manufacturers have produced suitable line drivers and receivers. Figure 9.22 gives

TABLE 9.6 EIA RS-232C electrical interface characteristic

CHARACTERISTIC	VALUE
Operating mode	Single ended
Maximum cable length	15 m
Maximum data rate	20 kilobaud
Driver maximum output voltage (open-circuit)	$-25\text{ V} < V < +25\text{ V}$
Driver minimum output voltage (loaded output)	$-25\text{ V} < V < -5\text{ V}$ or $+5\text{ V} < V < +25\text{ V}$
Driver minimum output resistance (power off)	300 Ω
Driver maximum output current (short-circuit)	500 mA
Maximum driver output slew rate	30 V/μs
Receiver input resistance	3–7 kΩ
Receiver input voltage	$-25\text{ V} < V_i < +25\text{ V}$
Receiver output state when input open-circuit	Mark (high)
Receiver maximum input threshold	-3 to $+3$ V

FIGURE 9.21 RS-232C interface



details of the 1488 quad RS-232C line driver and figure 9.23 gives details of the 1489 quad RS-232C line receiver. An example of the application of these chips is given in figure 9.24. Note that the 1489 receiver has an input control pin that can be used to define the amount of hysteresis at the input. We may leave this pin floating, in which case the input switching threshold is approximately 1 V.

RS-232C Interconnection Subset

So far we have seen three forms of RS-232C interconnection: the most basic arrangement of figure 9.8, the somewhat more complete circuit of figure 9.9, and the full RS-232C interface of table 9.5. A glance at table 9.5 makes it very clear that the RS-232C standard is aimed squarely at linking computer equipment with modems. Consequently, many of the facilities offered by the RS-232C standard are irrelevant to the engineer who wishes to connect a CRT terminal to a microcomputer.

Figure 9.25 shows a possible connection between two DTEs. As one DTE is the sink for the other's data, making the cross-connections shown in figure 9.25 is necessary. These cross-connections are frequently made at the junction of the cable and the connector. Such a cable linking a DTE to another DTE is called a *null modem* cable. When a DTE is connected to a modem (i.e., DCE) such a crossover is not necessary.

Many DTEs do not even use the subset of figure 9.25; for example, at the computer end of a computer-to-printer serial data link, no request to send (RTS) output may be provided. If the printer requires that its carrier detect input be driven by RTS, we need to strap the printer's carrier detect input to a logical one condition.

The 1987 Revision of RS-232C

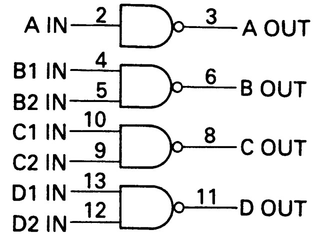
The RS-232C was updated in 1987 because it no longer represented current practice. Engineers were using it in ways not anticipated by those who drew up the original standard. RS-232C has now been replaced by EIA232D. The new standard is, as we

FIGURE 9.22 The 1488 quad line driver

FUNCTIONAL DESCRIPTION

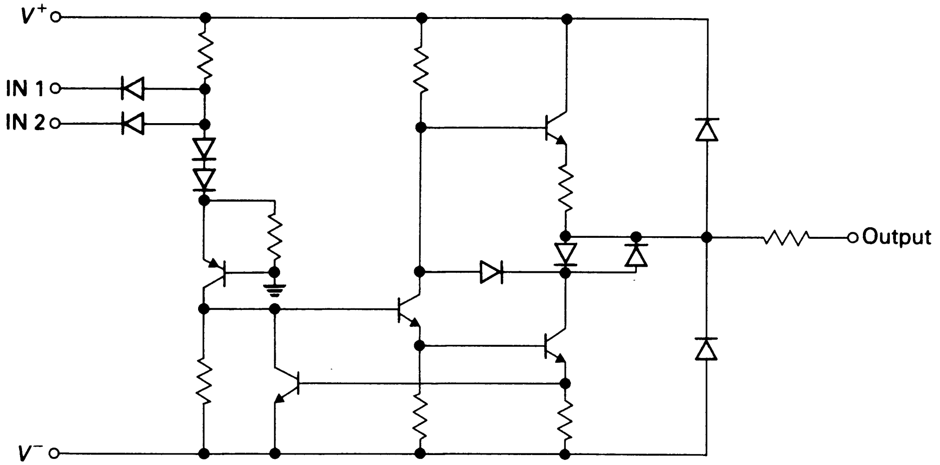
The 1488 is a quad line driver that conforms to EIA specification RS-232C. Each driver accepts one or two TTL/DTL inputs and produces a high-level logic signal on its output. The high and low logic levels on the output are defined by the positive and negative power supplies of plus and minus 9 volts, the output levels are guaranteed to meet the ± 6-volt specification with a 3 kΩ load. There is an internal 300 Ω resistor in series with the output to provide current limiting in both the high and low logic levels. The 1488 driver is intended for use with the 1489 or 1489A quad line receivers.

LOGIC SYMBOL

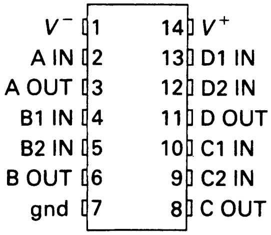


V^- = pin 1
V^+ = pin 14
gnd = pin 7

CIRCUIT DIAGRAM
(one driver shown)

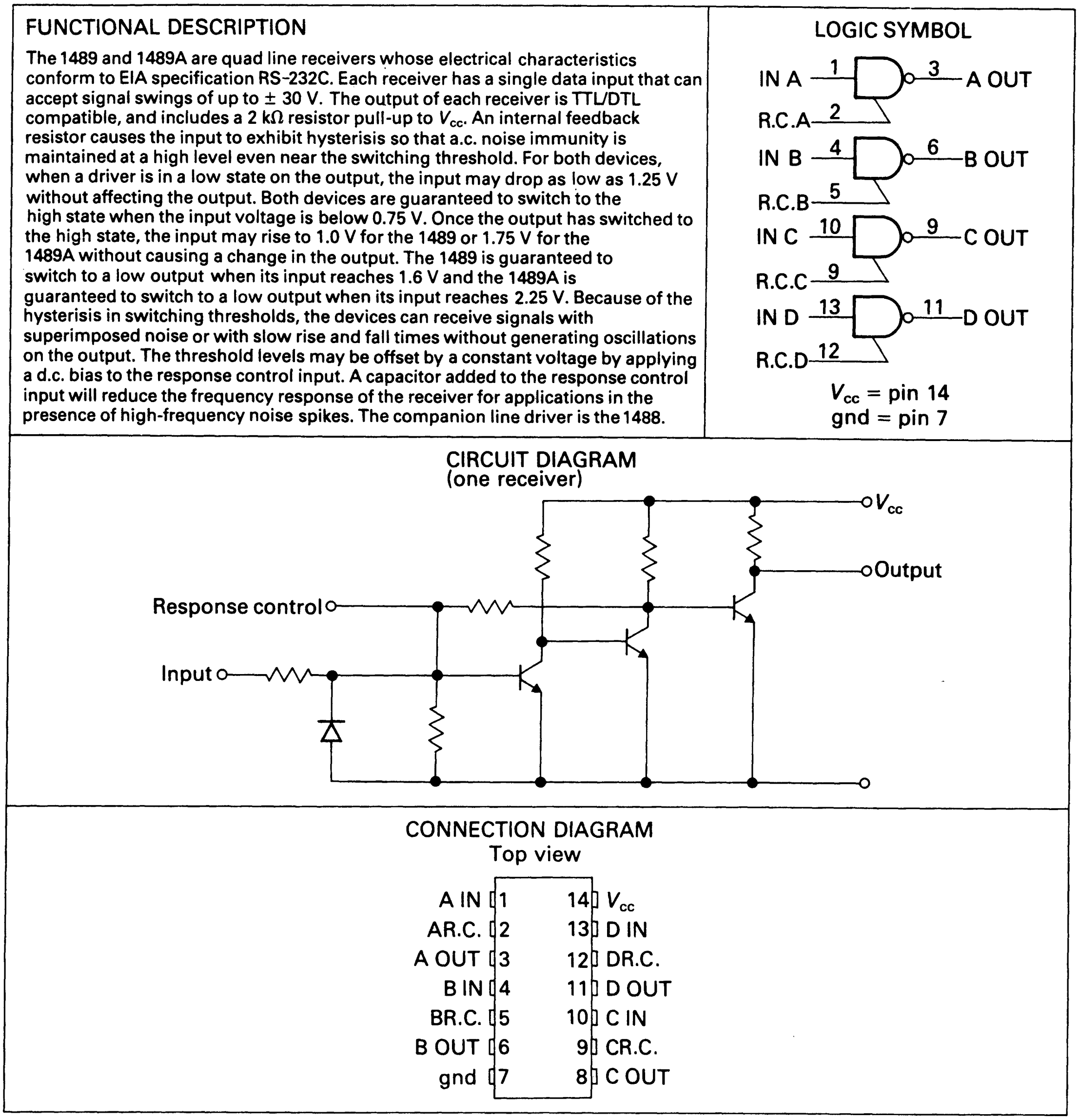


CONNECTION DIAGRAM
Top view



$V^- = -12$ V
$V^+ = +12$ V

FIGURE 9.23 The 1489 quad line receiver



FUNCTIONAL DESCRIPTION

The 1489 and 1489A are quad line receivers whose electrical characteristics conform to EIA specification RS-232C. Each receiver has a single data input that can accept signal swings of up to ± 30 V. The output of each receiver is TTL/DTL compatible, and includes a 2 kΩ resistor pull-up to V_{cc}. An internal feedback resistor causes the input to exhibit hysterisis so that a.c. noise immunity is maintained at a high level even near the switching threshold. For both devices, when a driver is in a low state on the output, the input may drop as low as 1.25 V without affecting the output. Both devices are guaranteed to switch to the high state when the input voltage is below 0.75 V. Once the output has switched to the high state, the input may rise to 1.0 V for the 1489 or 1.75 V for the 1489A without causing a change in the output. The 1489 is guaranteed to switch to a low output when its input reaches 1.6 V and the 1489A is guaranteed to switch to a low output when its input reaches 2.25 V. Because of the hysterisis in switching thresholds, the devices can receive signals with superimposed noise or with slow rise and fall times without generating oscillations on the output. The threshold levels may be offset by a constant voltage by applying a d.c. bias to the response control input. A capacitor added to the response control input will reduce the frequency response of the receiver for applications in the presence of high-frequency noise spikes. The companion line driver is the 1488.

LOGIC SYMBOL

V_{cc} = pin 14
gnd = pin 7

CIRCUIT DIAGRAM
(one receiver)

CONNECTION DIAGRAM
Top view

A IN	1	14	V_{cc}
AR.C.	2	13	D IN
A OUT	3	12	DR.C.
B IN	4	11	D OUT
BR.C.	5	10	C IN
B OUT	6	9	CR.C.
gnd	7	8	C OUT

might expect, compatible with the old standard, and only slight modifications have been made in order to match RS-232C more closely to its European equivalents CCITT V24 and V28 and to take account of actual current practice in linking DCEs to DTEs. The following are some of the changes to RS-232C:

1. The RS-232C pin 1, a protective ground, has been replaced by a shield. Pin 1 may be used to connect the screen of an interface cable to the frame of the

FIGURE 9.24 Example of an RS-232C data link

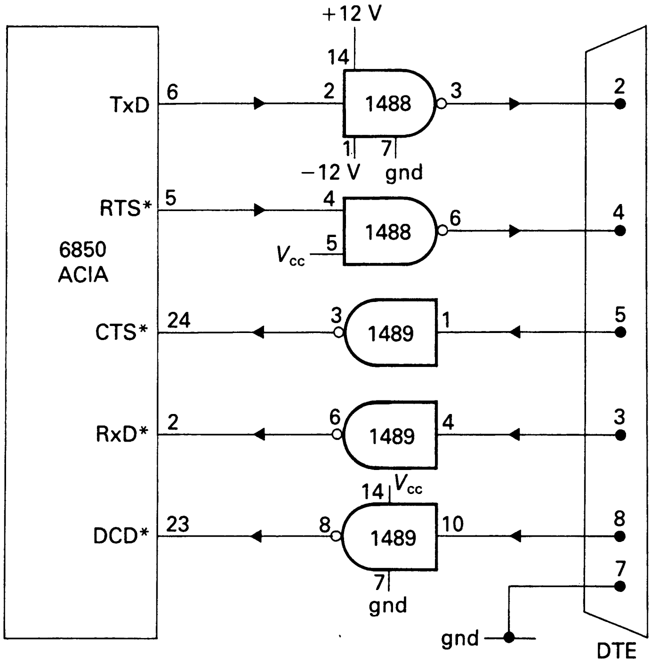
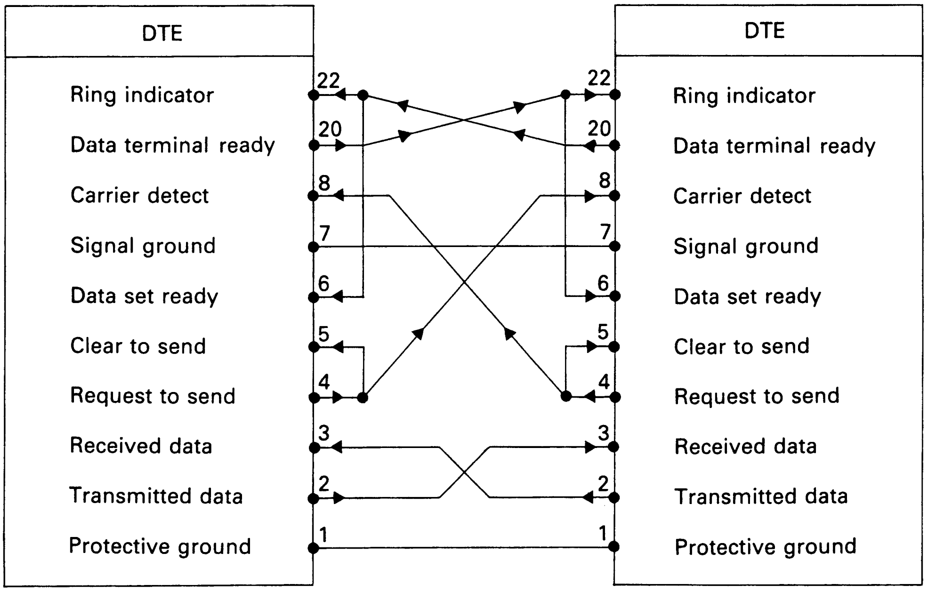


FIGURE 9.25 Connecting two DTEs together



DTE; that is, pin 1 is connected to the screen at only one end of the data link (this avoids ground-loop problems).

2. EIA232D now specifies the mechanical characteristics of the 25-pin interfaces. The old RS-232C specification only *recommended* the use of 25-pin D connectors in an appendix.

3. Provision for local and remote loopback testing has been made by defining three new signals (on pins 21, 18, and 25 of the D connector). Pin 21 is RL (remote loopback) and is asserted by the DTE to tell the local DCE to instruct the remote DCE to go into its loopback mode, allowing the local DTE to test both DCEs and the channel linking them; that is, the remote DCE at the other end of the data link will return signals received from the local DCE via the communication channel. Since the data is echoed back, it is very easy to test the operation of the data link by comparing the transmitted data with that echoed back. Modern peripherals such as the 68681 DUART provide automatic echo modes to facilitate testing. In the old RS-232C standard, pin 21 was a signal-quality detector used by the modem to indicate when a signal was of such a poor quality that it was no longer reliable.

Pin 18 in the new EIA232D standard is called LL (local loopback) and acts like pin 21, except that it establishes a loopback path through the local DCE only. Local loopback permits the system to be tested from the local CPU to the local DCE and back.

Pin 25 is called TM (test mode) and is asserted by the DCE to inform the DTE that the DCE is in a test mode because it has received either RL or LL from the local DTE or a message from the remote DCE requesting a test mode.

4. The recommendation that the RS-232C cable length be restricted to no more than 15 m (50 ft) has been removed; EIA232D permits longer transmission paths, whose length is determined by the electrical loading on the cable. One of the reasons for including this modification is that many users of the RS232C standard have been tolerating longer transmission paths than the legal maximum of 15 m.

The RS-422 and RS-423 Serial Interfaces

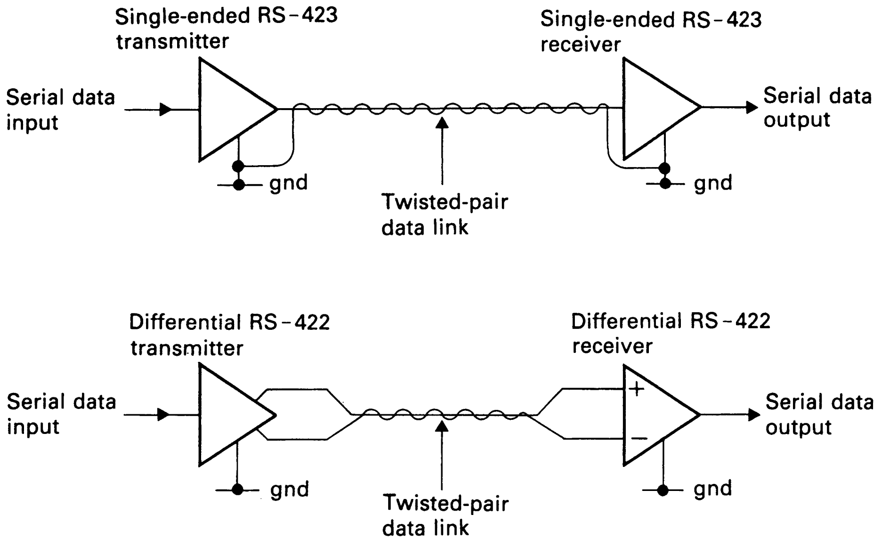
The RS-232C interface is now thought to be rather limited because of its low bandwidth and its maximum transmission path of only 15 m. Two improved standards for serial data links have been approved by the EIA. These standards are the RS-422 and RS-423, which define the electrical characteristics of a data link. Unlike RS-232C, these standards refer only to the electrical aspects of a data link.

Table 9.7 gives the basic electrical parameters of the RS-423 and RS-422 standards and figure 9.26 shows how they are arranged. The RS-423 standard differs little from the RS-232C standard of table 9.6. Indeed, the only real difference is that the RS-423 standard specifies much smaller receiver thresholds, permitting both a longer cable length and a higher signaling rate.

TABLE 9.7 EIA RS-422 and RS-423 electrical interfaces

CHARACTERISTIC	RS-423 VALUE	RS-422 VALUE
Operating mode	Single ended	Differential
Maximum cable length	700 m (2,000 ft)	1,300 m (4,000 ft)
Maximum data rate	300 kilobaud	10 megabaud
Driver maximum output voltage (open-circuit)	$-6\text{ V} < V_o < +6\text{ V}$	6 V between outputs
Driver minimum output voltage (loaded output)	$-3.6\text{ V} < V_o < +3.6\text{ V}$	2 V between outputs
Driver minimum output resistance (power off)	100 mA between −6 and +6 V	100 μA between +6 and −0.25 V
Driver maximum output short-circuit current	150 mA	150 mA
Maximum driver output slew rate	Determined by cable length and modulation rate	No limit on slew rate necessary
Receiver input resistance	>4 kΩ	>4 kΩ
Receiver maximum input voltage	$-25\text{ V} < V_i < +25\text{ V}$	−12 to +12 V
Receiver maximum input threshold	−0.2 to 0.2 V	−0.2 to +0.2 V

FIGURE 9.26 RS-432 and RS-422 serial interfaces



The RS-422 standard offers a significant improvement over both the RS-232C and RS-423 standards by adopting a balanced transmission mode. Balanced transmission requires two transmission paths per signal, because information is transmitted as a differential voltage between the two conductors. Noise voltages due to ground currents are introduced as common mode voltages that affect *both* transmission paths equally and have little effect on the differential signal between the lines. RS-422 systems can operate over distances of 15 m at 10 megabaud or over distances of 1300 m at 100 kilobaud.

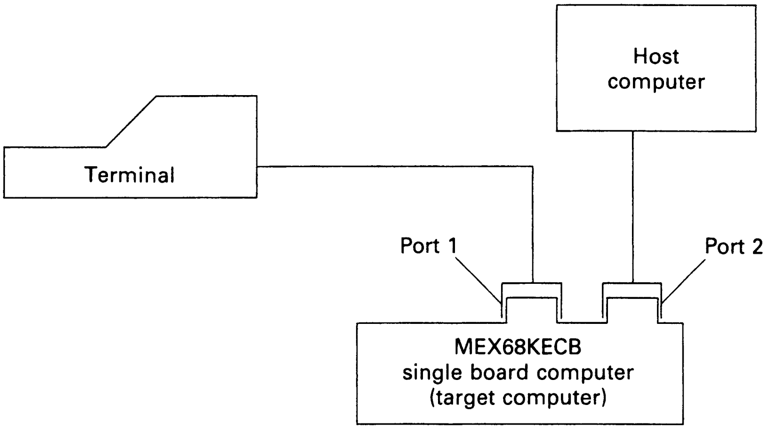
9.6 SERIAL INTERFACE FOR THE 68000

Instead of providing a relatively sterile textbook example of a 68000 serial interface based on the ACIA, examining the serial interface in Motorola's MEX68KECB single board microcomputer is most instructive. After all, this board is designed by real engineers to provide the maximum functionality while minimizing the board area taken up by the interface components and keeping their cost low.

Figure 9.27 shows how the ECB is arranged with respect to a terminal and a host computer. In a minimal mode, only the terminal interface on port 1 is necessary. This interface permits the user to interact with the ECB and to develop and debug software. The ECB has a parallel printer interface and an audio cassette interface, allowing programs and data to be printed and stored on tape, respectively. Unfortunately, the serial interface is rather slow and does not have any file-handling capability.

By connecting the ECB to a host computer via the ECB's second serial interface, port 2, a moderately powerful 68000 development system can be created. During my own initial 68000 development work, I used a 6809-based system running under the Flex 09 operating system as a host computer. After a reset, the

FIGURE 9.27 Relationship between the ECB, its console terminal, and a host computer



ECB communicates with the terminal through port 1. If the command TM ⟨exit character⟩ is entered, the ECB goes into its transparent mode. The expression ⟨exit character⟩ represents the character that must be entered from the terminal to leave the transparent mode. The default value is control A (ASCII $01).

Once the transparent mode has been entered, the terminal is effectively connected to port 2 and the 68000 on the ECB simply monitors the input from the terminal until it encounters the exit character. Therefore, the user can operate the host computer as if the ECB did not exist; for example, in the transparent mode I am able to edit a 68000 assembly language program on my host system and then assemble it into 68000 machine code using a cross-assembler. Once this process has been done, the exit character is entered and the terminal is once more "connected" to the ECB.

The next step is to transfer the machine code file on disk in the host system to the memory on the ECB. This step is performed by the load command, which moves object data in S-record format from an external device to the 68000's memory. S-record format is a way of representing machine code memory dumps. The syntax of the load command is

```
LO[port number][;⟨options⟩][=text].
```

Square brackets enclose options. Port number 2 (the default port number) specifies the host computer. Other options are not of interest here. The "[=text]" field is available only with port 2 and causes the text following the " = " to be sent to port 2 before the loading is carried out, thus allowing the user to communicate with the host computer. Suppose, for example, that we have a program on disk in the host system in S-record format whose file name is PROG23.BIN. To transfer this program to the 68000's memory space, we enter

```
LO2;=LIST PROG23.BIN
```

from the terminal, which sends the message "LIST PROG23.BIN" to the host computer. The message is interpreted by the operating system as meaning "list the file named PROG23.BIN." This file is then transmitted to port 2 and stored in the 68000's memory by its monitor software. When the loading is complete, the program can be executed and debugged using any of the ECB's facilities. Finally, the transparent mode may be entered and the source file, PROG23.TXT, reedited on the host system. This process is repeated until the software has been debugged.

Figure 9.28 gives the circuit diagram of the serial interface of port 1 and port 2 on the ECB. This diagram has been slightly simplified and redrawn from that appearing in the ECB user manual. Figure 9.28 includes both the serial interface between the ACIAs and the ports and the interface between the ACIAs and the 68000.

Interface between the ACIAs and the 68000 on the ECB

The CPU side of the ACIAs is fairly conventional as they are both connected directly to the 68000's data bus without additional buffering. Port 1 is connected to D_{08} to D_{15} and strobed by UDS*, whereas port 2 is connected to D_{00} to D_{07} and

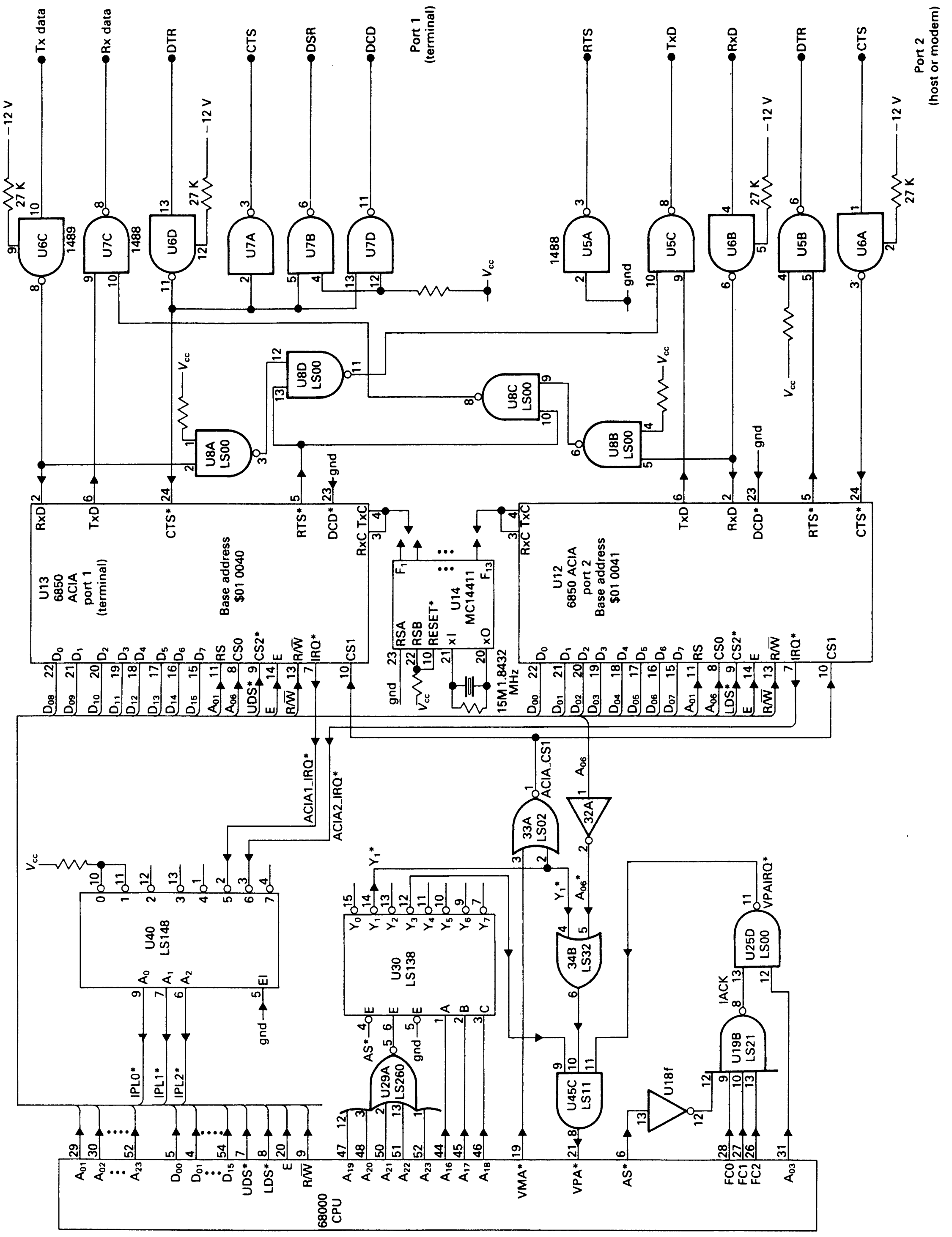


FIGURE 9.28 Circuit diagram of the ECB's serial interface. (Reprinted by permission of Motorola Limited)

strobed by LDS*. Address line A_{01} is connected to the register select input, RS, of each ACIA and is used to distinguish between the control/status and data register.

To simplify address decoding, each of the three ACIA's chip-select inputs are pressed into service: CS2* is connected to UDS* or LDS* to select between ACIAs, CS1 is connected to the output of the primary address decoding network, ACIA_CS1, and CS0 is connected to A_{06} from the CPU. Primary address decoding is performed by IC29a, a five-input NOR gate, and IC30, a three-line-to-eight-line decoder. These two chips decode A_{16} to A_{23} to produce an active-low signal, Y1*, from IC30. Y1* is combined with VMA* from the 68000 in a NOR gate, IC33a, to give the active-high ACIA_CS1 signal. Note that the ACIAs are not fully address decoded and take up the half of the 64K-byte page of memory space from $01 0000 to $01 FFFF for which A6 = 1.

Because the ACIA is interfaced to the 68000's synchronous bus, VPA* must be asserted whenever an ACIA is accessed. ICs 32a, 34b, and 45c perform this function. Note that VPA* is also asserted when the VPAIRQ* input to IC45c is asserted.

The final aspect of the interface between the ACIA and the CPU to be dealt with is the interrupt-handling hardware. Both ACIAs have independent interrupt request outputs. IRQ* from port 1 is wired to the level 5 input of a 74LS148 priority encoder (IC40) and IRQ* from port 2 is wired to the level 6 interrupt input.

During an interrupt acknowledge cycle, the output of the four-input AND gate IC19b goes active-high to generate an IACK signal. When a level 4 through 7 interrupt is acknowledged, A_{03} is high during the IACK cycle. Therefore, by combining A_{03} with IACK in IC25d, an active-low VPAIRQ* signal generated. VPAIRQ* is fed back to VPA* via IC45c, as indicated earlier. This arrangement converts interrupt levels 4 to 7 into autovectored interrupts, greatly simplifying the hardware design at the cost of reducing the number of possible interrupt vectors.

Serial Interface Side of the ACIAs

IC14, an MC14411 baud-rate generator, provides the transmitter and receiver clocks of the two ACIAs with a source of element timing at 16 times the baud rate of the transmitted or received signal elements. Jumpers on the ECB must be positioned to select the appropriate clock output from the MC14411 for both ACIAs. We should note that if the terminal is to communicate with the host computer, both ACIAs must operate at the same baud rate.

Little comment need be made about the connection of the ACIA's RS-232C signals to their respective ports. However, one interesting feature has been added to the ECB. Whenever the RTS* output of the port 1 (i.e., terminal) ACIA is asserted active-low, both ports 1 and 2 operate independently. However, whenever RTS* from ACIA1 is negated, ICs 6c, 8a, 8d, and 5c route the incoming data from port 1 to the outgoing data on port 2. Incoming data on port 2 is routed to port 1 via ICs 6b, 8b, 8c, and 7c when RTS* is negated. Consequently, negating RTS* connects port 1 to port 2. This is, of course, exactly what happens when the ECB enters its transparent mode. The software required to operate this type of serial link is described in detail in the section dealing with a monitor in chapter 11.

SUMMARY

In this chapter we have looked at the serial interface used to link digital systems to video display terminals and to modems. The simplest method of transmitting serial data is based on a character-oriented asynchronous protocol. If microprocessors had to perform the task of controlling serial links themselves, a considerable part of their power would be lost. We have examined the 6850 ACIA, which performs all serial to parallel and parallel to serial conversion itself. Once a 6850 ACIA is interfaced to a microprocessor, all the microprocessor has to do is to read data from or write data to the appropriate port. Moreover, the ACIA also checks the received data for both transmission and framing errors, further reducing the burden placed on the host microprocessor. Part of the power of the 6850 lies in its ability to cater for a wide variety of serial formats that are selectable under program control. The 6850 also provides three modem control signals, which further simplifies the design of a serial data link between a microprocessor system and a terminal or modem.

We have looked at the electrical interface between the serial transmission path and the microprocessor system. As the TTL-level voltages found in digital equipment are not best suited to transmission paths longer than a few meters, we have described the RS-232C, the RS-422, and the RS423 standards for the transmission of serial data over transmission paths that extend to 1000 meters or more.

Problems

1. What is the difference between asynchronous and synchronous transmission systems? What are the advantages and disadvantages of each mode of transmission?

2. What are the functions of the DCD*, CTS*, and RTS* pins of the 6850 ACIA?

3. The control register of a 6850 ACIA is loaded with the value $B5. Define the operating characteristics of the ACIA resulting from this value.

4. The status register of the 6850 is read and is found to contain $43. How is this value interpreted?

5. Write an exception-handling routine for a 6850 ACIA to deal with interrupt-driven input. Each new character received is placed in a 4K-byte circular buffer. Your answer must include schemes to (a) deal with buffer overflow and (b) deal with transmission errors.

6. Is connection of the output of an RS-423 transmitter to the input of an RS-232C receiver possible without violating the parameters of either standard? Is connection of an RS-232C output to an RS-423 input possible?

7. An asynchronous transmission system employs unsynchronized transmitter and receiver clocks, both of which are controlled by quartz crystals. It is guaranteed that the worst-case frequency difference between the clocks will never exceed .01 percent. A designer wishes to transmit long bursts of data asynchronously over a serial data link. Each data burst employs a start bit, a single parity bit, and a stop bit. What is the maximum permitted burst length if the designer caters for a maximum frequency error between transmitter and receiver clocks of 80

percent of the stated worst case? (The designer cannot employ the stated worst case value. Why?)

8. Define the following errors associated with asynchronous serial transmission systems and state how each might occur in practice:

a. Framing error b. Receiver overrun error
c. Parity error

9. Describe how bit-stuffing is employed by synchronous serial data links to ensure data transparency. Can you think of any other way in which data transparency can be achieved without resorting to bit-stuffing?

10. What are the advantages of the 68681 DUART over the 6850 ACIA?

11. Write a similar procedure to that of problem 5 for the 68681 DUART.

12. The DUART has a programmable baud-rate generator that is set by loading the appropriate value in clock-select register. This feature makes it possible to adapt to an "unknown" data rate. Write a subroutine that receives a string of carriage returns from a system (at an unknown speed) and adjusts the baud rate to match the incoming data. When the unknown baud rate has been determined, the DUART returns the string "ready."