PROJECT OBERON

The Design of an Operating System and Compiler

Niklaus Wirth
Jürg Gutknecht

Institut für Computersysteme
ETH Zentrum
8092 Zürich

ACM Press
New York, New York

Addison-Wesley Publishing Company
Wokingham, England • Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Amsterdam • Bonn • Sydney • Singapore
Tokyo • Madrid • San Juan • Milan • Paris • Mexico City • Seoul • Taipei
9 Device drivers

9.1 Overview

Device drivers are procedures that constitute the immediate interface between hardware and software. They refer to those parts of the computer hardware that are usually called peripheral. Computers typically contain a system bus that transmits data among its different parts. Processor and memory are considered as its internal parts; the remaining parts, such as disk, keyboard and display, are considered as external or peripheral, notwithstanding the fact that they are often contained in the same cabinet.

Such peripheral devices are connected to the system bus via special registers (data buffers) and transceivers (switches, buffers in the sense of digital electronics). These registers and transceivers are addressed by the processor in the same way as memory locations – they are said to be memory-mapped – and they constitute the hardware interface between processor bus and device. References to them are typically confined to specific driver procedures, which constitute the software interface.

Drivers are inherently hardware-specific, and the justification for their existence is precisely that they encapsulate these specifics and present to their clients an appropriate abstraction of the device. Evidently, this abstraction must still reflect the essential characteristics of the device, but not the details (such as the addresses of its interface registers).

Our justification to present the drivers connecting the Oberon System with the Ceres computer in detail is on the one hand the desire for completeness. But on the other hand, it is also in recognition of the fact that their design represents an essential part of the engineering task in building a system. This part may look trivial from a conceptual point of view; it certainly is not so in practice.

In order to reduce the number of interface types, standards have been established. The Ceres computer also uses such interface standards,
and we shall concentrate on them in the following presentations. The following devices are considered:

**Keyboard**

1. **Keyboard.** This is considered as a serial device delivering one byte of input data per key stroke. It is connected by a serial line according to the RS-232 and ASCII (American Standard Code for Information Interchange) standards. The software is contained in module Input (Section 9.2).

**Mouse**

2. **Mouse.** The Ceres mouse is a pointing device delivering coordinates in addition to three key states. For Ceres-1 and Ceres-2, the interface is non-standard; for Ceres-3, a serial transmission is used relying on the RS-232 standard. The software is part of module Input.

**Display**

3. **Display.** The interface to the display is an area of memory that contains the displayed information, one bit per pixel for the monochrome and four bits per pixel for a color display. The default size is 800 lines and 1024 dots per line. The software is module Display, which primarily consists of operations to draw frequently occurring patterns, so-called raster-ops (see Chapter 4).

**Disk**

4. **Disk.** The disk interface of Ceres-1 and Ceres-2 is non-standard and will not be described. The driver is contained in module Kernel. Ceres-3 operates without disk, but an optional hard disk can be connected through the standard SCSI interface described in Section 9.4.

**Diskette**

5. **Diskette.** The 3.5 in. diskette uses the same non-standard interface as the hard disk. Software: module Diskette. This interface is not described in this book.

**Serial line**

6. **Serial line.** This is the standard RS-232 serial interface allowing connections to be established between computers and to communicate over telephone lines via modems. The software interface is module V24 described in Section 9.2. Transmission rates go up to 19.2 Kbit/s.

**Network**

7. **Network.** Ceres computers may be connected by a local area network using the RS-485 standard. It operates with a transmission rate of 230 Kbit/s, and information is sent in packets of up to 512 bytes in the SDLC standard format. The interface software is module SCC described in Section 9.3; the interface hardware is a component called Serial Communications Controller.

**Real-time clock**

8. **Real-time clock.** A clock providing time and date is included in the Ceres computers and is used to record the creation time and date of files. Its interface is non-standard, hardly of general interest, and contained in module Kernel.

In all driver modules described below, procedures SYSTEM.PUT, SYSTEM.GET and SYSTEM.BIT are used to access the registers of the
device interface. Their first parameter is a (long) integer specifying the address of the register.

9.2 The RS-232 ASCII Standard for keyboard and serial line

All models of the Ceres computer are equipped with a component called a Universal Asynchronous Receiver and Transmitter (UART). It has an 8-bit parallel connection to the system bus, and two external connections – one for the transmitter and one for the receiver – resulting in a duplex transmission line. The UART performs the serialization of 8 bits upon sending and deserialization upon receiving. The 8 bits form a short packet, also called frame, and they are augmented by a start bit (always 1). There is no fixed time interval between consecutive packets. Transmission is called asynchronous because within the packet, there exists no explicit synchronization of clocks. The clock rates of the transmitter and the receiver must therefore be the same. The start bit is used to trigger the shift clock of the receiver. There is also a minimum time interval guaranteed between the last bit of a packet and the start bit of the next packet. It is measured in terms of bit times, and one may therefore think of a number of 'stop bits' as filling this interval. Finally, the packet may be augmented by a parity bit. The format of such a packet is shown in Figure 9.1.

The Ceres computer uses the Signetics 2692 UART, which contains two (almost) independent line interfaces called channels A and B. It also makes it possible to select several parameters, such as the transmission rate, the numbers of data bits and of stop bits, and the type of parity check (none, even, odd). The chosen values are stored in the UART’s parameter registers. The registers of primary importance are the data register and the status register.

When sending a byte, the processor must wait until the UART is ready. The ready state is asserted by bit 2 in the status register. Then the data byte is loaded into the data register, thereby automatically initiating transmission.

![Figure 9.1 ASCII character packet.](image-url)
For receiving a byte, one might use an analogous scheme. However, this would possibly introduce undesirable timing constraints and dependencies. We must consider the act of receiving a byte as part of the act of transmission, that is, as an action intimately connected with sending a byte. Sending and receiving must be performed at the same time. The UART itself provides some decoupling through its data buffers. However, they contain only a single byte (or a small number of them), and it is highly desirable to achieve greater decoupling.

This is possible by providing larger data buffers — typically on the side of the receiver — and by letting the main processor transfer a byte into that buffer as soon as it has been received. This requires that the processor be borrowed for a brief moment, which is achieved by a processor interrupt. No polling of the UART status for input is necessary in this case.

Channel A of the UART is used for the keyboard (input only). It is handled by module `Input`, whose interface is listed below:

```
DEFINITION Input;
  PROCEDURE Available(): INTEGER;
  PROCEDURE Read(VAR ch: CHAR);
  PROCEDURE Mouse(VAR keys: SET; VAR x, y: INTEGER);
  PROCEDURE SetMouseLimits(w, h: INTEGER);
  PROCEDURE Time(): LONGINT;
END Input.
```

The function procedure `Available` indicates the number of characters collected in the input buffer. If its value is greater than zero, `Read` delivers the next character (byte) from the buffered input stream. If no characters are available, `Read` implies a delay until a character has been received.

Module `Input` also contains the interface to the mouse (pointing device). The hardware contains two counters, one for the x direction and one for the y direction of movement. Procedure `Mouse` delivers the values of these counters and the state of the three buttons (keys). The latter are represented by the set element 0 for the right, 1 for the middle and 2 for the left button. `SetMouseLimits` permits determination of the limiting coordinate values of the rectangle into which the mouse position is mapped and where the cursor is drawn. The position ‘wraps around’ in both the horizontal and vertical directions. Changing the limits is useful when several displays are to be installed which are considered to lie in the same drawing plane, side by side.

The UART component also contains an additional counter — it is truly a multipurpose chip — that is incremented every 1/300 s and may serve for measuring elapsed time. The value of this counter is delivered by procedure `Time`. 
Device drivers

MODULE Input; (*NW 5.10.86 / 15.11.90 Ceres-2*)
IMPORT SYSTEM, Kernel;
CONST N = 32;
  MOUSE = 0FFFF8000H; UART = 0FFFFC000H; ICU = 0FFFF0000H;

VAR MW, MH: INTEGER; (*mouse limits*)
  T: LONGLONG; (*time counter*)
  n, in, out: INTEGER;
  buf: ARRAY N OF CHAR;

PROCEDURE Available();: INTEGER;
BEGIN RETURN n
END Available;

PROCEDURE Read(var ch: CHAR);
BEGIN
  REPEAT UNTIL n > 0;
  DEC(n); ch := buf[out]; out := (out + 1) MOD N
END Read;

PROCEDURE Mouse(var keys: SET; var x, y: INTEGER);
  var u: LONGLONG;
BEGIN
  SYSTEM.GET(MOUSE, u);
  keys := {0, 1, 2} = SYSTEM.VALSET, u DIV 10000H MOD 8;
  x := SHORT(u MOD 1000H) MOD MW;
  y := SHORT(u DIV 10000H) MOD 819;
  IF y >= MW THEN y := 0 END
END Mouse;

PROCEDURE SetMouseLimits(w, h: INTEGER);
BEGIN
  MW := w; MH := h
END SetMouseLimits;

PROCEDURE Time();: LONGLONG;
  var lo, hi: CHAR;
BEGIN
  REPEAT SYSTEM.GET(UART + 28, lo); SYSTEM.GET(UART + 24, hi);
  lo := T - LONG(ORD(hi)) - 256 = ORD(lo); SYSTEM.GET(UART + 28, lo)
  UNTIL lo1 = lo;
RETURN t
END Time;

PROCEDURE KBYTE;
  var ch: CHAR;
BEGIN
  SYSTEM.GET(UART + 12, ch); (*RHRA*)
  IF ch = 0FFX THEN HALT(24) END;
  IF n < N THEN buf[in] := ch; in := (in + 1) MOD N; INC(n) END
END KBYTE;

PROCEDURE CTint;
  var dmy: CHAR;
BEGIN
  SYSTEM.GET(UART + 60, dmy); (*stop timer*)
  INCT, OFFFFH), SYSTEM.GET(UART + 56, dmy) (*restart timer*)
END CTint;
BEGIN MW := 1024; MH := 800;
 n := 0; in := 0; out := 0; T := OFFFF;
 Kernel.InstallIP(KBINT, 4); Kernel.InstallIP(CTInt, 0);
 SYSTEM.PUT(UART+16, 10X); (*ACR*)
 SYSTEM.PUT(UART+ 8, 15X); (*CRA enable*)
 SYSTEM.PUT(UART, 13X); (*MR1A, Rxdy –Int, no parity, 8 bits*)
 SYSTEM.PUT(UART, 7X); (*MR2A 1 stop bit*)
 SYSTEM.PUT(UART+ 4, 44X); (*CSRA, rate = 300 bps*)
 SYSTEM.PUT(UART+52, 14X); (*OPCR OP4 = KB and OP3 = CT int*)
 SYSTEM.PUT(UART+28, 0FFX); (*CTRL*)
 SYSTEM.PUT(UART+24, 0FFX); (*CTUR*)
 SYSTEM.GET(UART+56, buf[0]); (*start timer*)
 SYSTEM.PUT(ICU + 4, 18X); (*clear ICU IMR and IRR bits 0*)
 SYSTEM.PUT(ICU + 4, 3CX); (*clear ICU IMR and IRR bits 4*)
END Input

Comments
(1) The fact that many properties of the UART can be parametrized
leads to a longer initialization sequence. We refrain from
explaining all the details specific to the Signetics 2692 part and
refer to the pertinent device specifications. Here it may suffice
to state that the RS-232 transmission for the keyboard uses a rate
of 300 bit/s, 8 bits without parity check, and 1 stop bit.

(2) An interrupt handler is declared in Oberon as a parameterless
procedure marked with a plus sign. The keyboard interrupt
handler KBINT receives a single character. If it is the abort
character (ctrl-shift-delete), a trap is induced by the statement
HALT(24). This enables the operator to interrupt a computation
when it appears to be non-terminating.

(3) The keyboard buffer is designed as a circular buffer. When full,
incoming characters are ignored (except abort).

(4) The UART’s timer (counter) has 16 bits and is accessed in two
steps, reading the high and low half respectively. A 32 bit
extension is provided in the form of variable T, which is
incremented by 216 upon each timer interrupt, which occurs when
the UART-counter t has reached zero. Since the UART counter
is decremented every 1/300 s, the value delivered by procedure
Time is computed as T – t.

(5) Before an interrupt channel can be active, three conditions must
be satisfied:
   (i) the processor interrupt must be enabled – it normally is;
   (ii) the interrupt mechanism of the device must be enabled;
   (iii) the interrupt control unit (ICU) lying between devices and
processor must let the respective interrupt signal pass.

The third condition is established by the last two statements of
the module’s initialization sequence, one instruction being
necessary for the keyboard’s interrupt, one for the timer’s.
(6) The mouse counter values serve as coordinates for the cursor to be displayed on the screen. For this reason, they are confined to the ranges $0 \leq x < MW$ and $0 \leq y < MH$. The limit values can be set by procedure *SetMouseLimits* according to the dimension (resolution) of the available display.

Channel B of the UART leads to an external RS-232 (V24) connector. This serial line is not employed by the Oberon system and hence is freely available to the user. Its software interface is module *V24*.

```plaintext
DEFINITION V24:
IMPORT SYSTEM;
PROCEDURE Start(CSR, MR1, MR2: CHAR);
PROCEDURE SetOP(s: SET);
PROCEDURE ClearOP(s: SET);
PROCEDURE IP(x: INTEGER): BOOLEAN;
PROCEDURE SR(n: INTEGER): BOOLEAN;
PROCEDURE Available(): INTEGER;
PROCEDURE receiver(VAR x: SYSTEM.BYTE);
PROCEDURE Send(x: SYSTEM.BYTE);
PROCEDURE Break;
PROCEDURE Stop;
END V24.
```

The channel’s receiver and transmitter are started by calling procedure *Start*. It has three parameters, whose values are codes for the transmission clock rate, the parity mode, the number of bits per byte and for the number of stop bits. Procedure *Available* denotes the number of bytes available (received) in the input buffer. Procedure *Receive* delivers the next byte of the sequence, and *Send* dispatches the byte specified by the parameter. *SR* yields the value of the specified bit in the status register, and *Stop* serves to turn off both transmitter and receiver.

The Signetics 2692 dual UART’s exaggerated multipurpose nature becomes apparent at this point: in addition to the two channels and a counter/timer, it also contains an input and an output register (7 and 8 bits respectively) with external connections (pins). On the Ceres computer, these signals are used in various ways and must not be used by the programmer, except the following:

- **Available**
- **Receive**
- **Send**
- **Input port**
- **Output port**

**Modem signals**

<table>
<thead>
<tr>
<th>input:</th>
<th>output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 DCD</td>
<td>0 DTR</td>
</tr>
<tr>
<td>1 CTS</td>
<td>1 RTS</td>
</tr>
<tr>
<td>2 DSR</td>
<td></td>
</tr>
</tbody>
</table>

The meaning of these bits is derived from their use in connection with modems. Procedures *SetOP* and *ClearOP* serve to set and clear those
Table 9.1 UART Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RHRA/B</td>
<td>receiver holding register</td>
</tr>
<tr>
<td>THRA/B</td>
<td>transmitter holding reg.</td>
</tr>
<tr>
<td>MR1A/B</td>
<td>mode register 1</td>
</tr>
<tr>
<td>MR2A/B</td>
<td>mode register 2</td>
</tr>
<tr>
<td>CRA/B</td>
<td>command register</td>
</tr>
<tr>
<td>CSRA/B</td>
<td>clock select register</td>
</tr>
<tr>
<td>SRA/B</td>
<td>status register</td>
</tr>
<tr>
<td>OPCR</td>
<td>output port configuration register</td>
</tr>
<tr>
<td>IPCR</td>
<td>input port change register</td>
</tr>
<tr>
<td>ISR</td>
<td>interrupt status register</td>
</tr>
<tr>
<td>IMR</td>
<td>interrupt mask register</td>
</tr>
<tr>
<td>CTUR</td>
<td>counter/timer upper byte value</td>
</tr>
<tr>
<td>CTLR</td>
<td>counter/timer lower byte value</td>
</tr>
</tbody>
</table>

bits of the UART's output register (OP) that are specified in their set parameter. Function procedure IP serves to test the specified bit in the UART's input register (IP). And finally, procedure Break serves to apply a break signal (0 value during at least 20 ms) to the serial line. Table 9.1 lists the principal registers of the UART.

MODULE V24; (*NW 18.3.69 / 19.1.91*)
("interrupt-driven UART channel B")
IMPORT SYSTEM, Kernel;

CONST BufLen = 512;
UART = 0FFFFCOOH; ICU = 0FFFF9000H;

VAR in, out: INTEGER;
buf: ARRAY BufLen OF SYSTEM.BYTE;

PROCEDURE+ Int;
BEGIN SYSTEM.GET(UART+44, buf[in]); in := (in+1) MOD BufLen
END Int;

PROCEDURE Start*(CSR, MR1, MR2: CHAR);
BEGIN in := 0; out := 0; Kernel.InstallP(Int, 2);
SYSTEM.PUT(UART+40, 30X); (*CRB reset transmitter*)
SYSTEM.PUT(UART+40, 20X); (*CRB reset receiver*)
SYSTEM.PUT(UART+36, CSR); (*CSR8 clock rate*)
SYSTEM.PUT(UART+40, 15X); (*CRB enable Tx and Rx, pointer to MR1*)
SYSTEM.PUT(UART+32, MR1); (*MR1B, parity, not bits*)
SYSTEM.PUT(UART+32, MR2); (*MR2B stop bits*)
SYSTEM.PUT(UART+20, 20X); (*IMR RxRdy Int enable*)
SYSTEM.PUT(ICU + 4, 1AX); (*ICU IMR and IRR bit 2*)
END Start;
PROCEDURE SetOP*(s: SET);
BEGIN SYSTEM.PUT(UART + 56, s)
END SetOP;

PROCEDURE ClearOP*(s: SET);
BEGIN SYSTEM.PUT(UART + 60, s)
END ClearOP;

PROCEDURE IP*(n: INTEGER): BOOLEAN;
BEGIN RETURN SYSTEM.BIT(UART + 52, n)
END IP;

PROCEDURE SR*(n: INTEGER): BOOLEAN;
BEGIN RETURN SYSTEM.BIT(UART + 36, n)
END SR;

PROCEDURE Available*(): INTEGER;
BEGIN RETURN (in - out) MOD BufLen
END Available;

PROCEDURE Receive*(VAR x: SYSTEM.BYTE);
BEGIN
  REPEAT UNTIL in # out;
  x := buf[out]; out := (out + 1) MOD BufLen
END Receive;

PROCEDURE Send*(x: SYSTEM.BYTE);
BEGIN
  REPEAT UNTIL SYSTEM.BIT(UART + 36, 2);
  SYSTEM.PUT(UART + 44, x)
END Send;

PROCEDURE Break*;
VAR i: LONGINT;
BEGIN SYSTEM.PUT(UART + 40, 60X); i := 50000;
  REPEAT DEC(i) UNTIL i = 0;
  SYSTEM.PUT(UART + 40, 70X)
END Break;

PROCEDURE Stop*;
BEGIN SYSTEM.PUT(UART + 20, 0); (*IMR disable Rx-Int*)
  SYSTEM.PUT( ICU + 4, 3AX) (*ICU chan 2")
END Stop;

END V24.

9.3 The RS-485 SDLC Standard for a network

The Ceres-Oberon System also features a network connection. The principal differences between the RS-232 line and the network connection are that of a point-to-point line versus a bus with multiple taps, and
9.3 The RS-485 SDLC Standard for a network 265

that of asynchronous versus synchronous transmission. Asynchronous
transmission as described in the preceding section is uneconomical if
transmission speeds beyond 20 Kbit/s are desired, because too much
time is wasted between consecutive bytes. Synchronous transmission
improves performance, and it is used by the Ceres computers for
interconnection in a local area network.

Strictly speaking, the difference between so-called asynchronous
and synchronous transmission lies in the packet length only, because
the former uses synchronicity during the transmission of each byte,
too. The price for longer packets lies in the need for more accurate
clocks; clock accuracy limits packet length, unless some encoding scheme
is used to transmit the clock together with data. The RS-485 and SDLC
Standards do not specify such an encoding; the clock is not transmitted.
It is fixed here to 230 Kbit/s, yielding about 30 Kbyte/s.

A direct consequence is that computation of the byte sequence
and transmission of the packet cannot be interleaved due to the strict
timing constraints. Since one byte must be sent every 30 μs, the data
of the entire packet must be ready before transmission is initiated.

The SDLC (Synchronous Data Link Control) standard specifies a
fixed packet format of variable length. The role of the start bit is taken
by a start byte, a so-called flag. It is followed by the data bytes, and
the packet is terminated by another flag. The flag consists of 6
consecutive ‘1’s. Hence, any occurrence of 6 consecutive ‘1’s must not
occur within the data section. The problem is solved by the transmitter
automatically inserting and the receiver removing a zero bit after every
occurrence of five consecutive ones. If the bit following the 5 ones is
not a zero, a flag was received. This zero insertion (and deletion), as
well as the pre- and postfixing of a flag, is performed automatically by
the interface hardware component SCC (Zilog 8530).

In order to detect transmission errors, the transmitter computes
a cyclic redundancy code (CRC) over the data and appends it to the
data stream, just before the terminating flag. The receiver computes the
same code and compares it with the received code. If the difference is
not zero, a status bit is set that must be inspected for each received
packet.

The SDLC Standard also requires that the first byte of the packet
– the one after the flag – specify the receiver’s address. This is necessary,
because in a network the recipient is not automatically determined like
in a point-to-point connection. Every station is therefore given a unique
identification. Beyond this, we postulate some additional properties of
packets. Each packet consist of a header followed by the data. The first
9 bytes constitute the header, of which the first is the destination
address, the second denotes the sender’s address, and the third a packet
type. It is followed by two bytes indicating the packet length (in bytes).
(The remaining four bytes are currently not used.) The resulting packet format is shown in Figure 9.2, and it is reflected by the data type Header.

**DEFINITION SCC;**

```
TYPE Header =
    RECORD valid: BOOLEAN; dadr, sadr, typ: SHORTINT;
        len: INTEGER; (*of data following header*)
        destLink, srcLink: INTEGER (*unused*)
    END;
```

PROCEDURE Start(filter: BOOLEAN);
PROCEDURE Send(VAR head: Header; VAR buf: ARRAY OF SYSTEM.BYTE);
PROCEDURE Available(): INTEGER;
PROCEDURE ReceiveHead(VAR head: Header);
PROCEDURE Receive(VAR x: SYSTEM.BYTE);
PROCEDURE Skip(m: INTEGER);
PROCEDURE Stop;
END SCC.

As in the case of the V24 interface, the receiver buffers the incoming data stream (without flags and CRC). Procedure Receive picks consecutive bytes from the buffer. The number of buffered bytes is given by procedure Available. The task of receiving is simplified by procedure ReceiveHead. It is called when a next packet is expected. The field valid has the meaning 'packet has been received and header is valid'.

Transmitter and receiver can be switched on and off by calls to procedures Start and Stop. The former features a Boolean parameter filter with the meaning 'filter out packets that are not addressed to this station'. The interface chip is capable of comparing the first header byte (destination address) with its own station's address (stored in a register), and to discard a packet upon mismatch. An active filter is of course the
normal mode of operation, because in this mode discarded packets do not require interaction with the computer’s processor.

The SCC-driver program is, as one expects, dominated by accesses to the device’s registers. The primary function of these registers is indicated in Table 9.2; for further details, we refer to the controller’s data sheets. It must suffice to say that registers (except the data register) are accessed in two steps. First, the register’s number is sent to the control port, and thereafter its value is transferred. (Register number and value are time-multiplexed; see procedure PUT.) Table 9.2 gives an overview of the available device registers.

MODULE SCC; (*NW 13.11.87 / 22.8.90 Ceres-2*)
IMPORT SYSTEM, Kernel;

CONST BufLen = 2048;
com = 0FFFFD008H; (*commands and status, SCC channel A*)
dat = 0FFFFD00CH;
DIPS = 0FFFFC001H;
ICU = 0FFFFE004H;
RxCA = 0; (*R0: Rx Char Available*)
TxBE = 2; (*R0: Tx Buffer Empty*)
Device drivers

Hunt = 4; (*R0: Sync/Hunt*)
TxUR = 6; (*R0: Tx UnderRun*)
RxOR = 5; (*R1: Rx OverRun*)
CRC = 6; (*R1: CRC error*)
EOF = 7; (*R1: End Of Frame*)

TYPE Header =
  RECORD valid*: BOOLEAN;
    dest*: addr*, typ*: SHORTINT;
    len*: INTEGER; (*of data following header*)
    destLink*, srcLink*: INTEGER (*link numbers*)
  END;

VAR in, out: INTEGER;
  Addr: SHORTINT;
  SCCR3: CHAR;
  buf: ARRAY BufLen OF CHAR;

PROCEDURE PUT(r: SHORTINT; x: SYSTEM.BYTE);
BEGIN SYSTEM.PUT(com, r); SYSTEM.PUT(com, x)
END PUT;

PROCEDURE Int;
  VAR del, oldin: INTEGER; stat: SET; dmy: CHAR;
BEGIN SYSTEM.GET(dat, buf[in]);
  PUT(1, 0X); (*disable interrupts*)
  oldin := in; in := (in+1) MOD BufLen; del := 16;
  LOOP
    IF SYSTEM.BIT(com, RxCA) THEN del := 16;
    IF in # out THEN SYSTEM.GET(dat, buf[in]); in := (in+1) MOD BufLen
    ELSE SYSTEM.GET(dat, dmy)
  END
  ELSE SYSTEM.PUT(com, 1X); DEC(del);
    IF SYSTEM.BIT(com, EOF) & (del <= 0) OR (del <= -16) THEN EXIT END
  END;
  SYSTEM.PUT(com, 1X); SYSTEM.GET(com, stat);
  IF (RxOR IN stat) OR (CRC IN stat) OR (in = out) THEN
    in := oldin (*reset buffer*)
    ELSE in := (in-2) MOD BufLen (*remove CRC*)
  END;
  SYSTEM.PUT(com, 30X); (*error reset*)
  SYSTEM.PUT(com, 10X); (*reset ext/stat interrupts*)
  PUT(1, 8X); (*enable Rx-Int on 1st char*)
  SYSTEM.PUT(com, 20X); (*enable Rx-Int on next char*)
  PUT(3, SCCR3); (*enter hunt mode*)
END Int;

PROCEDURE Start (*filter: BOOLEAN);
BEGIN in := 0; out := 0;
  IF filter THEN SCCR3 := 0D00 ELSE SCCR3 := 0D9X END;
  SYSTEM.GET(DIPS, Addr); Addr := Addr MOD 40H;
  Kernel.Install(*Int1, 1);
  PUT(9, 80X); (*reset A, disable all interrupts*)
  PUT(4, 20X); (*SDLC mode*)
  PUT(1, 0X); (*disable all interrupts*)
  PUT(2, 0X); (*interrupt vector*)
9.3 The RS-485 SDLC Standard for a network

PROCEDURE SendPacket(varargin head, buf: ARRAY OF SYSTEM.BYTE); VAR 4, len: INTEGER;
BEGIN head[2] := Adr;
len := ORD(head[5]) + length head[4]);
LOOP "sample line" i := 60;
  REPEAT DEC(i) UNTIL SYSTEM.BIT(com, Hunt) OR (i = 0);
  IF i > 0 THEN "line idle" END;
  i := i + 128 + 800; "delay"
  REPEAT DEC(i) UNTIL i = 0
END;
 KernelSetICU(0A2X); "disable interrupts!"
PUT(5, 63X); "RTS, send 1s"
PUT(5, 68X); "RTS, Tx enable"
SYSTEM.PUT(com, 80X); "reset Tx-CRC"
SYSTEM.PUT(dat, ORD(head[1])); "send dest"
SYSTEM.PUT(com, 0C0X); "reset underrun/EOM flag"
REPEAT UNTIL SYSTEM.BIT(com, TxBE);
i := 2;
REPEAT SYSTEM.PUT(dat, head[i]); INC(i);
  REPEAT UNTIL SYSTEM.BIT(com, TxBE)
UNTIL i = 10;
i := 0;
WHILE i < len DO
  SYSTEM.PUT(buf[i]); INC(i); "send data"
  REPEAT UNTIL SYSTEM.BIT(com, TxBE)
END;
REPEAT UNTIL SYSTEM.BIT(com, TxUB) & SYSTEM.BIT(com, TxBE);
PUT(5, 63X); "RTS, Tx disable, send 1s"
i := 300;
REPEAT DEC(i) UNTIL i = 0;
PUT(5, 0E1X); "~RTS"
PUT(1, 8X); "enable Rx-Int on 1st char"
PUT(14, 21X); "enter search mode"
SYSTEM.PUT(com, 20X); "enable Rx-Int on next char"
PUT(3, SCCR3); "enter hunt mode"
Device drivers

PROCEDURE Available(t: INTEGER);
BEGIN
RETURN (in - out) MOD BufLen
END Available;

PROCEDURE Receive(x: SYSTEM.BYTE);
BEGIN
REPEAT UNTIL in # out;
x := buf[out]; out := (out+1) MOD BufLen
END Receive;

PROCEDURE ReceiveHead(head: ARRAY OF SYSTEM.BYTE);
BEGIN
IF (in - out) MOD BufLen >= 9 THEN head[0] := 1; i := 1;
REPEAT Receive(head[i]); INC(i) UNTIL i = 10
ELSE head[0] := 0
END
END ReceiveHead;

PROCEDURE Skip(m: INTEGER);
BEGIN
IF m <= (in - out) MOD BufLen THEN
out := (out+m) MOD BufLen ELSE out := in
END
END Skip;

PROCEDURE Stop;
BEGIN
PUT(89, 0X); (reset SCCA)
SYSTEM.PUT( ICU, 39X); SYSTEM.PUT( ICU, 99X); (reset IMR and IR8)
END Stop;

BEGIN Start(TRUE)
END SCC.

Comments

(1) The sequence in which individual registers are initialized is essential for correct functioning. The lack of its specification in the part's documentation was a source of serious difficulties and headaches.

Timing considerations

(2) Some sections of the driver are very time-critical (particularly for Ceres-I). For example, after receiving an interrupt, the first byte must be read immediately. A queue of 3 bytes in the receiver hardware allows for enough time to disable interrupts, store the byte and copy the buffer index (oldin), which is used as a reset point in case of a transmission error.

(3) Before sending a packet, it must be verified that the line is free by testing the so-called hunt bit. If the line is busy, the line is
9.4 A disk driver using the SCSI interface 271

Collision detection

pollled again after a delay. The delay is influenced by the station's address, causing all stations to have a slightly different delay. Actual collisions can only be detected by the receiver through the CRC check at the end of the packet.

(4) After transmitting the last data byte, the line must be kept busy for transmitting CRC and flag, and for the receiver to terminate (stop bits!). This time span is of order 200 μs, which is too short for the timers' resolution, and hence must be programmed as a tight delay loop. The delay constant depends on the computer's clock rate and model. This is rather unfortunate.

(5) The end of a packet is indicated by the EOF bit in the SCC's status register. Unfortunately, it is not reliable. It sometimes signals the end prematurely. The situation is saved by testing a number of times while no further data bytes arrive. The resulting program section does not appear to be very neatly conceived – but software that fixes a hardware deficiency never does.

(6) Procedure Skip serves to discard received data, namely the next $m$ bytes. $m$ is compared with the stored number of bytes given by $n = (\text{in-out}) \mod \text{BufLen}$, in order not to 'overshoot'. We leave any reader so inclined to discover why the guard $m \leq n$ is correct, whereas $m < n$ would be wrong.