Amélie serial communications

Preliminary specification - 2006/11/12

Introduction

In order to simplify serial communications, the BIOS makes it easier for applications to interface with a remote computer by serial means.

Two buffers are implemented:

1. **128** byte *receive* buffer
2. **96** byte *transmit* buffer

In order to send serial data, you should call the BIOS routine, passing a byte or a pointer to a null-terminated string (as appropriate).
In order to receive serial data, hook into the *SERIAL_RXSTRING* vector and/or the *SERIAL_RXBYTE* vector.

There are no methods for controlling the serial format. The BIOS sets up `9600bps, 8N1` during system reset. If you should desire something different, you will need to poke the ACIA yourself.

Byte suppression

In *normal* use, all ASCII 13 codes are discarded. This is done because RISC OS uses <10> as a line separator, while DOS/Windows systems use <10><13>. By omitting all <13>s, you do not need to modify your comms software to communicate with Amélie.

Binary mode
In *binary* mode, selected by writing a byte in Page Zero, all <13>s will be passed into the receive buffer, and the *SERIAL_RXSTRING* vector is never called.

The BIOS interface

This describes the BIOS interface to the serial system. This is what you need to know in order to use the serial system.

There are seven calls. Three to receive, two to send, and two status calls. In order to use the serial system, you should set `A` to the entry code, set `X` and `Y` (as applicable), and then `JSR &FFE2`.
It should be assumed that registers are corrupted. Where the entry/exit conditions do not specify a specific register, you may assume that the contents are unimportant (entry) or undefined (exit).

SERIAL_ISACTIVE (`&FFE2, entry 0`)
This lets you know whether or not serial communications are available.
On entry:
`A = 0`
On exit:
`X =` *Zero* if *no* serial comms, else *non-zero*.
`Y =` Transmit buffer bytes free.

SERIAL_DISABLE (`&FFE2, entry 1`)
This allows you to forcibly disable serial communications when they are enabled. Note that the user can restore communications by deasserting, then reasserting, the `DCD` line.
On entry:
`A = 1`
`X = 0`
On exit:
Registers undefined.
This call will *not* fail if serial is already disabled, or not available.

SERIAL_GETBYTE (&FFE2, entry 2)
This returns the *next* byte in the serial buffer.
On entry:
A = 2
On exit:
X = Byte
Y = Is *zero* if *no* byte available, else *non-zero*.

SERIAL_PEEKBYTE (&FFE2, entry 3)
This returns the *next* byte in the serial buffer *without* altering the read pointer.
On entry:
A = 3
On exit:
X = Byte
Y = Is *zero* if *no* byte available, else *non-zero*.

SERIAL_GETSTRING (&FFE2, entry 4)
This copies an entire chunk of the serial buffer in to a nominated address. The space that you provide does not need to be any larger than 128 bytes, and it *cannot* span a page boundary.
Note that the BIOS calls the *SERIAL_RXSTRING* vector upon receiving ASCII 10. Therefore the string returned will be terminated by ASCII 10. It is valid for the entire string returned to be a single ASCII 10.
On entry:
A = 4
X = High byte (page) of your buffer address
Y = Low byte (offset) of your buffer address
On exit:
Y = Is *zero* if *no* data available, else *non-zero*.

SERIAL_SENDBYTE (&FFE2, entry 5)
This adds a single byte to the output buffer.
On entry:
A = 5
X = Byte to send
On exit:
X = Is *zero* if transmit buffer overrun, else *non-zero*.
Y = Is *zero* if *no* serial available, else *non-zero*.

SERIAL_SENDSTRING (&FFE2, entry 6)
This copies a *null*-terminated string into the serial buffer. The location of the string *cannot* span a page boundary.
On entry:
A = 6
X = High byte (page) of your buffer address
Y = Low byte (offset) of your buffer address
On exit:
X = Is *zero* if transmit buffer overrun, else *non-zero*.
Y = Is *zero* if *no* serial available, else *non-zero*.

Buffer behaviour

It is important to realise a few things about the serial buffering.

Receive buffer
This is a buffer of 128 bytes. There are two pointers, a *head* and a *tail*. The *head* points to the location of the *next* write offset, while the tail points to the location of the *next* read.

On the following page is an explanation of the *head* and *tail* pointers in action…

```
Serial buffer at &xx00:
     &xx00     R    <-- tail
     &xx01     i
     &xx02     c
     &xx03     k
     &xx04          <-- head
```

In the example above, the BIOS has received "Rick", but the application code has not read anything. Upon reading, when the *tail* meets the *head*, we can assume the buffer has been cleared and the pointers are both reset to zero.

In order not to swamp the buffer, use is made of the CTS line. When 128 bytes have been received, CTS is deasserted. If further data comes in (i.e. the remote system is ignoring the handshake), that subsequent data will be silently discarded - as this will be taken as a failing of the remote system and not of Amélie.
In addition to this, CTS is deasserted when the BIOS calls the *SERIAL_RXSTRING* vector. CTS will then be reasserted when the application calls a serial read routine. The reason for this is so that, upon receiving a string, the application can copy it out (and the buffer then cleared) without worrying about subsequent data being appended after the string. Data subsequently received *will* be appended, but this - again - is taken as a fault of the host for ignoring the handshake.

Transmit buffer
This is a buffer of 96 bytes. The application code may write strings and bytes to this buffer (the *head* pointer). The data contained in the buffer is written out under interrupt (the *tail* pointer). As for the receive buffer, if the *head* and *tail* pointers equal, they are reset to zero as that will mean the buffer has been sent.
If there are no serial communications, or it is disabled, then *all data sent to the buffer is silently discarded* (and the Y register set to zero on return). This may sound bizarre, however it is to allow in-line debugging data and/or status messages without worrying about the overheads of "is serial available? is it ready?". Just put bytes in the transmit buffer - if there is serial, they'll be sent. If not, they'll be tossed.

If the transmit buffer should exceed 96 bytes, then *the entire buffer contents will be tossed*. You may use the *SERIAL_ISACTIVE* routine to read the number of bytes free in the transmit buffer. If this eventuality should come to pass, the X register will be zero on return.

Internal serial handling

This section is *not* necessary for using the serial system. It is here for those who would like a deeper explanation of the serial system.

The ACIA causes an interrupt on one of three conditions - a change of state on the DCD line, a character received, and a character transmitted. There is a fourth interrupt, a change of state on the DSR line, but this is not implemented.

* **Change of DCD state**
  If DCD has been asserted, we can flag serial as being active, else it is not connected. Amélie provides a link to jumper DTR to DCD at the serial connection for devices that don't set DCD themselves.

* **Serial byte received (normal mode)**
  If the serial buffer is *not* full, the byte is read and placed in the receive buffer, *unless* the byte was ASCII 13 in which case it is silently discarded.
  If the receive buffer is now full, serial activity is suspended (to deassert CTS). Serial activity will not be resumed until the buffer has been cleared.
  If the byte received was ASCII 10, then the *SERIAL_RXLINE* vector will be called, else the *SERIAL_RXBYTE* vector will be called.
  There is no case handling for data received when CTS is not asserted as it appears from the 6551 datasheet that the receiver is inactive in this state.

* **Serial byte received (binary mode)**
  This is largely the same as normal mode reception - so I shall just describe the differences. It is actually the *same* code performing both implementations, however I felt adding this to the above would just clutter things.
  The byte read from the ACIA is copied to the receive buffer. *All* bytes are copied, including ASCII 13. Accordingly, the BIOS does not ever call the *RXLINE* vector, only *SERIAL_RXBYTE* for each byte received.

* **Serial byte transmitted**
  When a byte is written to the transmit buffer, if the *tail* pointer is zero then the first byte is copied to the ACIA to start off the transmit process.
  Then, upon receiving a "byte sent" IRQ, the transmit buffer will be examined. If there is another byte to send (i.e. *head* pointer is not zero), then the byte is copied to the ACIA and the *tail* pointer incremented. If *head* and *tail* now match, the buffer has been entirely transmitted and thus the pointers will be reset. Upon the final "byte sent" IRQ, the BIOS will see that the *head* pointer is zero so no further action will be taken.

As you can see, the BIOS takes care of serial communications. It does so by way of the *ACIAIRQ* vector, so you could - if required - replace the serial system.
The BIOS sets the serial communications protocol to 9600 baud, 8 data bits, 1 stop bit, no parity. It sets this *once* on system reset. If you require something else, you can poke the hardware directly for, perhaps, 2400 7E1?

Rick, 2006/11/12
heyrick *-at-* merseymail *-dot-* com
http://www.heyrick.co.uk/amelie/