USLOSS Supplement - Terminals

NOTE:
This document does not give any information that is not already present in the USLOSS documentation. However, this document will (hopefully) present that information in a form that is easier to understand - especially for people without much experience accessing hardware devices.

1 Overview

Each terminal in USLOSS is a very simple device, that has the ability to send and receive only one character at a time in each direction[1]. The 4 terminals, however, run entirely in parallel; there is no interaction between them, and no need to share state across them. Thus, each terminal can be reading and writing entirely independently of the others.

Each terminal is managed by two registers: the Control Register, and the Status Register. The Control Register is used by the CPU to send commands to the device, and the Status Register is used by the device to send information back to the operating system. These two registers are addressed using the same device type and unit number; the only difference is that when you call `DeviceInput()`, the CPU is reading the Status Register, and when you call `DeviceOutput()`, the CPU is writing to the Control Register. Thus it is **impossible** for the CPU to read the Control Register, or to write to the Status Register.

2 Interrupts

Each terminal will send an interrupt to the CPU each time that it needs attention; however, these interrupts start off masked, and thus the CPU will not be disturbed.

Masking off device interrupts is entirely **unrelated** to whether interrupts are enabled or disabled inside the CPU. For example, if interrupts are unmasked inside the device but disabled inside the CPU, then the device will send interrupts to the CPU - but these interrupts will be deferred. The current code will continue to run, entirely undisturbed, until interrupts are re-enabled inside the processor; once they are re-enabled, the interrupt from the terminal device will then be delivered, immediately. Similarly, if interrupts are enabled inside the CPU but are masked off inside the terminal device, then the terminal will not send any interrupts to the CPU, even if it needs attention.

[1] That is, a terminal can be simultaneously be writing a single character, and also reading a different character - but it can never send two, or receive two, at the same time

Each terminal device has two masks: one for reading, and another for writing. If the read interrupt is unmasked, then the terminal will interrupt the processor each time that it has read a new character from input, and this character is ready for the CPU to read. If the read interrupt is unmasked, then the terminal will interrupt the processor each time that the current output character is flushed out, and the terminal is ready to start writing another.

2.1 How to Change Masks

Interrupt masks may be changed by writing to the Control Register. This register is also used for writing characters to output, so programmers must be careful how they use it:

- When altering the interrupt masks, the programmer must be careful to not accidentally sent a character to output. (**Pay attention to this - I've seen multiple students make this mistake.**)
- When sending a character to output, the programmer must be careful to not accidentally change the interrupt masks.

The format of the Control Register can be found in the USLOSS documentation, page 9. You will notice three control bits (0,1,2) in the register; when changing interrupt masks, make sure to set "Send Char" to zero, so as to not send a character as output.

2.2 Init

When USLOSS begins, all terminal interrupts are masked off. Most operating systems will probably want to enable interrupts almost immediately - at least, the Recv (that is, reading) interrupt - so that the OS is responsive to input.

It is permissible to also turn on the "Xmit" (that is, write) interrupt in init as well; however, it is also permissible, if preferred, to only turn this interrupt on while some process is attempting to write to the terminal.

2.3 Handling Interrupts

While there are two different reasons why a terminal might send an interrupt - and a separate mask bit in the Control Register for each - each terminal device only has one interrupt; the CPU does not know, at first, whether the interrupt was for the read side or the write side (or perhaps both). To find this out, the interrupt handler must read the Status Register, and check the status fields for each side.

> **WARNING:**
> Reading the Status Register will, if a character is available to be read, consume that register immediately; it may not be available in the future. Thus, it is important to never read the Status Register

without checking for a character; a given character will probably be visible only once.[2]

When an interrupt occurs, and you read the status using `DeviceInput()`, you will first look at two fields in the Status Register (USLOSS documentation, page 9). Each status field is 2 bits in size, and will be one of three values:

- `USLOSS_DEV_READY`

  For Read, this means that no new character is available on input (yet).

  For the Write/Xmit side, this means that the previous output character (if any) has been flushed to the terminl, and the device is ready to be given another character to write. (Of course, writing out another character will require you to write to the Control Register; unlike the Read, this operation has not happened yet.)

- `USLOSS_DEV_BUSY`

  For the Read/Recv side, this means that another character has been read from input; the top 8 bits of the Status Register contain the character that was read. **Do not read the Status Register a second time; the character is in the current Status.**

  For Write, this means that a character is still being written to output.

  You must not start another Write, while the terminal is still busy with a previous one; however, since the Read and Write channels are entirely independent, it is perfectly normal to start another Write even while the Read side is still busy, and vice-versa.

- `USLOSS_DEV_ERROR` means that there is some sort of problem; you should not hit this if your system is working properly - so you may want to `Halt(1)` the simulation if this occurs.

USEFUL MACROS

The macro `USLOSS_TERM_STAT_RECV(x)` and `USLOSS_TERM_STAT_XMIT(x)` will read the Recv and Xmit status fields out of a Status register.

The macro `USLOSS_TERM_STAT_CHAR(x)` will read the character (that is, the top 8 bits) out of a status register.

(Of course, these macros don't actually **perfrom** the read of the Status Register; you must do that yourself. But they make it easier to interpret this register's contents.)

(document continues on the next page)

[2]The USLOSS documentation doesn't explicitly say this, but it appears to be true. And this sort of design is definitely common in Real World hardware.

3 Example: Writing to the Control Register

In the snippet below, we write to the Control Register, to send a new character. In this example, we choose to have both the Read and Write interrupts unmasked.

```
int   unit = ... ;    // which terminal to write to
char *buf  = ... ;    // a string in memory, somewhere
int   i    = ... ;    // the index, into the string, of the next char to send

int cr_val = 0x1;     // this turns on the 'send char' bit (USLOSS spec page 9)
cr_val    |= 0x2;     // recv int enable
cr_val    |= 0x4;     // xmit int enable
cr_val    |= (buf[i] << 8);    // the character to send

USLOSS_DeviceOutput(USLOSS_TERM_DEV, unit, (void*)(long)cr_val);
```

4 Strategies for Reading and Writing

Assuming that you are unwilling to poll the device, it will be difficult to write a simple loop, for either reading or writing, which directly interacts with the device. This is because, as noted above, the act of reading the Status Register necessarily consumes any character on the Read side. Therefore, we recommend that your interrupt handler (or, device driver directly triggered by the interrupt) always read the Status Register exactly **once.** It should check the status on both the Recv and Xmit sides, and call one (or both, based on what is available:

```
void handle_one_terminal_interrupt(int unit, int status)
{
    if ( ... recv is ready ... )
    {
        ... read character from status ...
        ... place character in buffer ...
        ... wake up waiting process (if appropriate) ...
    }

    if ( ... xmit is ready ... )
    {
        if ( ... a previous send has now completed a "write" op ... )
            ... wake up a waiting process ...

        if ( ... some buffer is waiting to be flushed ... )
            ... send a single character ...
    }
}
```