University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

EECS 150 **Spring 2002**

Checkpoint : MIDI Interface

This lab is an official project checkpoint. It is to be completed with a project partner on your assigned project board. Project checkpoints are designed to keep you on a weekly schedule towards completion of the project. In each week's lab, you will be asked to check-off some significant step in the design.

This lab marks a transition from what we think of as "spoon-feeding" to "hand-holding." From now in the semester, you will be given you less information on how to do the lab work. You will be making your own design decisions and figuring out your own details. You are encouraged to discuss the design problems with TAs and other students.

1 Objectives

The objective of this checkpoint is to demonstrate the ability to recognize MIDI commands sent from a MIDI keyboard. This demonstration consists of two milestones in the development of your project:

1. Correct electrical connection from the MIDI receptacle on the Xilinx board to the FPGA, utilizing an optoisolator.
2. Correct functioning of a MIDI parser.

The "MIDI parser" refers to a circuit that examines input bytes from a MIDI source and looks for particular patterns. This project requires a MIDI parser to recognize *note-on* and *note-off* commands. All other commands **must** be ignored (except as part of adding extra functionality to one's design). Refer to the project specification for details on MIDI commands and on MIDI hardware interfacing. Remember that there are two different ways to encode note-off commands. Additionally, be sure to read the Addendum on dealing with real-time messages and running status included at the end of this document.

2 Lab Assignment

First, you should read and understand the project specification. If you have any questions about the project spec, get them cleared up early.

The physical MIDI interface requires wiring-up the Motorola MOC5009 optoisolator (or the Sharp PC900V/PC900VQ) and associated discretes pack (2 resistors and 1 diode). These components should be connected up as per Figure 6 of the project specification. Note that the 5-pin MIDI interface in the figure is drawn as if looking into the male end of a MIDI cable (not the female DIN connection on the board). The circuit will not function if the interface is wired in a mirror image.

This project checkpoint requires three circuit designs for the FPGA:

1. A UART transceiver, as designed in Lab #7.

2. A MIDI parser that takes input from the UART and recognizes MIDI note-on and note-off commands. The brains of this circuit are a simple FSM. This is a good chance to try one-hot encoding for your state machine. An output interface for the parser is suggested in the project specification and shown below (NOTE-ON and NOTE-OFF status bits which pulse when a command is received, plus KEY and VELOCITY registers to denote command contents).

3. A MIDI client that takes input from the parser and displays information about recognized commands. The NOTE-ON/NOTE-OFF, KEY and VELOCITY values produced by the parser can be displayed one at a time on the 7-segment LEDs. You may use a pair of SW5 DIP-switches to select which value to display (check the pinout sheets to see which SW5 DIP switches are still available)

Demonstrate your MIDI interface by connecting it to a MIDI keyboard in the lab and playing a few notes. Try individual notes as well as several notes at a time.

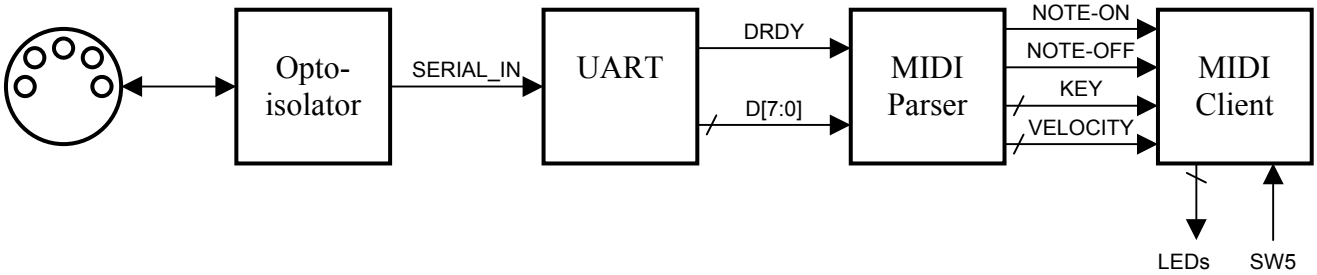

Figure 1: MIDI interface with suggested I/O

3 Acknowledgements

Original lab by J. Wawrzynek (Fall 1994).
Modifications by N. Weaver, E. Caspi, J Sampson.

Name: _____ Name: _____ Lab: _____

4 Check-offs
(MIDI Interface)

Pre-lab

- Wire wrap optoisolator + discretes. TA: _____ (10%)

Lab Assignment

- Working optoisolator + discretes (visualize output on oscilloscope) TA: _____ (20%)
- Complete MIDI interface TA: _____ (70%)

Completed on time TA: _____ (×100%)

Completed 1 week late TA: _____ (×50%)

ADDENDUM ON RUNNING STATUS AND REAL-TIME MESSAGES

To support a larger variety of MIDI keyboards (our gray ones and yours from home), you will need to extend your MIDI parser to support the "running status" convention.

What is "running status?"

From "The USENET MIDI Primer",
<http://www.harmony-central.com/MIDI/Doc/primer.txt>

For all of these messages, a convention called the "running status byte" may be used. If the transmitter wishes to send another message of the same type on the same channel, thus the same status byte, the status byte need not be resent.

This convention allows a transmitter to compress the data stream by dropping status bytes. A command without a status byte implicitly uses whatever status byte was most recently sent. For example, playing a triad chord (3 notes together) may be encoded as a sequence of note-on commands, only the first of which has a status byte:

```
1001nnnn 0kkkkkkk 0vvvvvvv   0kkkkkkk 0vvvvvvv   0kkkkkkk 0vvvvvvv
^^^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^
(1st note)                    (2nd note)           (3rd note)
```

Similarly, the pressing and releasing of a key may be encoded as a note-on followed by a note-on with 0-volume and no status byte:

```
1001nnnn 0kkkkkkk 0vvvvvvv   0kkkkkkk 00000000
^^^^^^^^^^^^^^^^^^^^^^^^^^   ^^^^^^^^^^^^^^^^^
        (key press)              (release)
```

To support running status, your MIDI parser must be extended to do the following:

(1) remember the most recent status byte (store in a register).
(2) process a command beginning with a data byte
    if the most recent status byte was a note-on or note-off

(3) ignore a command beginning with a data byte
    if the most recent status byte was not a note-on or note-off.

If your parser was designed as a traditional state machine then the addition of running-status is relatively simple. You probably have a starting state that begins processing a command when it receives a note-on or note-off status byte. That state must now also respond to a data byte, either jumping ahead to process the new command (case (2) above) or ignoring the byte (case (3) above). If your parser is not a traditional FSM, you may need to be more ingenious. Alternatively, you may find it easier to just redesign the parser as an FSM.

Status and data bytes are easy to differentiate. Status bytes have an MSB (most significant bit) of 1, while data bytes have an MSB of 0.

There is one additional complication of which you should be aware but which we will handle for you. This is the "real-time messages," a collection of single-byte messages which may appear in the middle of other messages. Since we plan to ignore such messages, the easiest way to deal with them is to filter them out of the MIDI byte stream before they reach the parser. The parser can then remain blissfully unaware of pesky real-time messages.

From "The USENET MIDI Primer",
<http://www.harmony-central.com/MIDI/Doc/primer.txt>

REAL TIME MESSAGES.

This is the final group of status bytes, 0xf8 - 0xff. These bytes are reserved for messages which are called "real-time" messages because they are allowed to be sent ANYPLACE. This includes in between data bytes of other messages. A receiver is supposed to be able to receive and process (or ignore) these messages and resume collection of the remaining data bytes for the message which was in progress. Realtime messages do not affect the "running status byte" which might be in effect. All of these messages have no data bytes following (or they could get interrupted themselves, obviously). The messages are as follows:

0xf8	timing clock
0xf9	undefined
0xfa	start
0xfb	continue
0xfc	stop
0xfd	undefined
0xfe	active sensing
0xff	system reset

The above description suggests that the following are all valid ways to transmit note-on messages interspersed with "active sensing" bytes: (status 0xFE)

1001nnnn [0xFE] 0kkkkkkk 0vvvvvvv

1001nnnn 0kkkkkkk [0xFE] 0vvvvvvv

1001nnnn 0kkkkkkk 0vvvvvvv [0xFE] 0kkkkkkk 0vvvvvvv

Note the use of running status in the last example.

Your project needs to know about real-time messages for two reasons. First, many MIDI keyboards, including our gray ones, send periodic "active sensing" messages (status 0xFE). Second, MIDI sequencers, which play-back pre-recorded music, send timing messages (status 0xF8,0xFA,0xFB,0xFC). We may use a sequencer later in the semester to test your project with a pre-recorded MIDI message stream. Although

your project needs to implement only the note-on and note-off commands, it must be smart enough to not be confused by real-time messages.

Addendum by E. Caspi (Spring 2000). Edited by J. Sampson.