FEATURE ARTICLE

Bill Dudley

Build a MIDI Sustain Pedal

Bill set out to take care of the shortcomings of his wife's electronic keyboard by building a MIDI sustain pedal using a 68HC11 and C. Sit back and enjoy the performance as he gets the hardware and software in tune.

This was one of those projects that sounds like a good idea then quickly turns into spending so much time building the widget that you could buy ten commercially manufactured widgets for the money you didn't make while building the project. But, it was more fun than breaking rocks for a living.

My wife is a musician, a keyboard player. One of her recent acquisitions was a small, lightweight keyboard. This keyboard was cheap so it didn't provide for a sustain pedal.

Initially, Kate thought it would be OK, but after a while, she decided that a sustain pedal would be nice. That's when I spoke up.

"Sure honey, I can build you a sustain pedal. (pause) What's a sustain pedal, anyway?" And that's how this project started.

A sustain pedal, when pressed, removes the damping mechanism from the sound-producing mechanism, so a note will play until it decays away naturally. In a piano, the felt damping pads are lifted so the strings do not stop oscillating when the key is released.

MIDI has been the subject of several articles ("Digital Attenuators," *Circuit Cellar* 95; Jeff's MIDI series in *Circuit Cellar* 99–100), so I won't spend time describing it, except to say that MIDI stands for Musical Instrument Digital Interface and defines a protocol and physical medium for interconnecting musical input devices (keyboards, mostly) and musical output devices (namely, synthesizers).

MIDI uses current loop as the physical medium, the bit rate is 31,250 bps, and the messages consist of packets of (typically) 1–3 bytes.

I designed and built this project using some fairly high-powered tools and hardware to minimize development time. Obviously, if this design was going to be mass produced, you could spend more time in the design phase and cram the design into a tiny little microprocessor (e.g., a PIC).

Because I was making only one unit, I optimized for short design time. In this article, I describe the development process to show how designs are done in larger companies with reasonable tool budgets.

HARDWARE

First, I chose a microprocessor. I toyed with the idea of using an 8048 or a PIC, but since they have no internal serial port, I rejected them.

The MIDI data rate of 31,250 bps is high enough so the timing would be tight without a serial port to divide the interrupt rate by eight. I also had to deal with simultaneous input and output streams, so a bit-banging serial

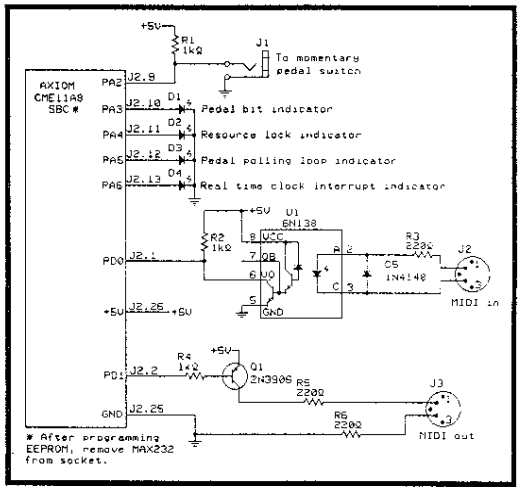

Figure 1—*This is all the custom I/O you need to convert the 68HC11's serial port to MIDI. Because the MIDI interface is a current loop, the MAX232 supplied on the Axion SBC is removed when the MIDI interface is used.*

Listing 1—*The `midi()` task reads incoming MIDI messages and copies them out again, locking the MIDI-out resources when a MIDI message is partially processed. The header files whose names begin with a "c" (`cclock.h`, etc.) are generated by the RTXCGEN tool supplied with the RTXC kernel.*

```
#include "hardware.h"
#if K4
#include <iok4.h>
#else
#include <io.h>
#endif

#include "rtxcapi.h"

#include "cclock.h"   /* CLKTICK */
#include "cres.h"     /* SCIRES  */
#include "cqueue.h"
#include "csema.h"
#include "lcd.h"

#define SELFTASK ((TASK)0)
#define TMINT ((TICKS)5000/CLKTICK)

static const char hex[] =
{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
  'A', 'B', 'C', 'D', 'E', 'F' };

void xtoa(unsigned int x, int len, char *s) {
  s[len--] = '\0';
  while(len >= 0) {
    s[len--] = hex[x & 0x000f];
    x >>= 4;
  }
}

char buf[4];
extern char unsigned minbuf[], ihead, itail;
extern unsigned char porta;
static unsigned int inseq;

void midi(void)
{
unsigned char true, ichar, havelock;
  true = 1;
  havelock = inseq = 0;
  init_lcd();
  gotoxy_lcd(1, 1);
  cputs_lcd("ON ");
  gotoxy_lcd(1, 3);
  cputs_lcd("OFF ");
  gotoxy_lcd(1, 1);
  while(true) {
    KS_wait(SCIISEMA);              /* wait on input char */
    while (itail != ihead) {
      ichar = minbuf[itail++];
      switch (ichar & 0xf0) {
        case 0x90 :
          buf[0] = '\0';
          inseq = 3;
          gotoxy_lcd(4, 1);
          break;
        case 0x80 :
          buf[0] = '\0';
          inseq = 3;
          gotoxy_lcd(5, 3);
          break;
        case 0xa0 :
        case 0xb0 :
        case 0xe0 :
          inseq = 3;
          gotoxy_lcd(1, 2);
          goto Print;
        case 0xc0 :
        case 0xd0 :
          inseq = 2;
          gotoxy_lcd(1, 2);
          goto Print;
        case 0xf0 :
          switch(ichar) {
            case 0xf0 :             /* system exclusive */
              inseq = 0xffff;
```

(continued)

port would have to handle interrupts at a 62-kHz rate.

I've had a lot of experience with the 6811 [1] and a little experience with the 8051, so I went with what I know. The only argument against the 6811 is that it's probably overkill for this project, but remember, I'm optimizing for development time, not cost.

I've used the 68HC11 C0 and K4 variants on professional jobs, but they have too much I/O capability for this project. Besides, mail-order 68HC11 SBCs always use the A or the E part. Good enough, we'll use the 68HC11A.

After searching the 'Net and looking through the ads in *Circuit Cellar*, I decided on Axiom's CME11A SBC. It has a 68HC11A microprocessor, with a MAX232 and DE-9 hung off the serial port, sockets for RAM/ROM/EEPROM, onboard power-supply regulator, plus an I/O decoder and header pins to connect an LCD module and keyboard.

Axiom also sells a development package that includes a wall-wart power supply, serial-port cable to connect to a PC, and software to compile and download programs to the onboard EEPROM.

This package is handy for quick demonstrations or prototypes. There's even a wire-wrap area on the CME-11A so you can add some custom I/O.

The Axiom board's built-in LCD port makes it convenient to attach an LCD module [2] that uses the Hitachi 44780 or equivalent LCD controller. This setup allows a convenient debug message display from your embedded system, which is useful if the target's only serial port is otherwise occupied.

TOOLS

When I'm building an embedded system, I always start in C. If I run out of room or real time, I'll drop back into assembler, but generally that isn't necessary. ROM is just too cheap.

(Whenever I've started an embedded project in industry, I've always designed in the current "popular" ROM size, with a way to expand to the "cutting edge" size ROMs. Every time, without exception, by the time the project hits production, the ROM I designed in is almost obsolete, and the "big" ROM is the default size,

and there's a much bigger one available that won't fit in the socket I designed. You'd think I'd have learned Moore's law by now.)

The C compiler I use is from Cosmic Software. It generates excellent code and comes with a reasonable subset of the standard C library.

There's a command-line version and a GUI version. I'm strictly a command-line kind of person. I just want to type in my C code, type `make`, and go get a Jolt.

`Make` is portable across all the systems I use (Berkeley Unix, Linux, DOS). So, I don't have to change the way I work even as I move from computer to computer during the day.

The next wonderful tool is an emulator. The crash-and-burn development cycle can get tedious, especially when nothing's running so you can't even run a debugger on the target.

An emulator lets you quickly see that your stack pointer is pointing to nowhere. Then, once you get the little sucker running, the emulator provides a profiler, instruction trace, complex breakpoints, and the ability to examine or change the registers or memory.

My Nohau emulator lives in a box the size of a toaster, communicates with a PC running DOS/Windows, and has a cable ending in a pod which plugs into the target in place of the microprocessor chip. Nohau-supplied software runs under Windows and lets you control the emulator. You can load code, set breakpoints, examine or change registers, and run your code.

This emulator has real-time full-speed trace of the target's execution and a performance analyzer to help figure out where the code is spending its time.

SOFTWARE

I decided this project needed an RTOS so I could have multiple tasks running independently without writing this complexity myself. I went with RTXC from Embedded Systems Products. It comes with complete source code and is customized for the particular microprocessor you're using.

The RTXC kernel supplies all needed (and even imagined) kernel services—that is, manipulation of mailboxes, semaphores, resource locks, queues, dynamic tasks, and so on.

Listing 1—*continued*

```
            gotoxy_lcd(1, 4);
            goto Print;
          case 0xf2 :   /* song pos */
            inseq = 2;
            gotoxy_lcd(1, 4);
            goto Print;
          case 0xf3 :   /* song sel */
          case 0xf6 :   /* tune req */
          case 0xf7 :   /* end of sys excl */
          case 0xf8 :   /* timing clock */
          case 0xfa :   /* start */
          case 0xfb :   /* continue */
          case 0xfc :   /* stop */
          case 0xfe :   /* active sensing */
          case 0xff :   /* reset */
            default :   /* undefined: f1 f4 f5 f9 fd */
              inseq = 1;
              gotoxy_lcd(1, 4);
              goto Print;
            }
          default :
Print:
          xtoa((unsigned int)ichar, 2, buf);
          buf[2] = ' ';
          buf[3] = '\0';
          break;
          }
        cputs_lcd(buf);
        if(!havelock) {
        KS_lockw(SCIRES);
        havelock = 1;
      }
      porta |= SCIRESBIT;
      PORTA = porta;
      KS_enqueuew(SCIOQ, &ichar);
      if(inseq) inseq--;
      if(inseq == 0) {
      porta &= ~SCIRESBIT;
      PORTA = porta;
      KS_unlock(SCIRES);
      havelock = 0;
    }
  }
 }
}
```

One of the nice parts about this tool set is that the three vendors—Cosmic, Nohau, and Embedded Systems Products—communicate with each other. For example, the emulator knows about the symbol table output by the compiler. The compiler works correctly with the kernel. And the kernel's internal tables and variables are understood by the emulator.

THE TASKS

This project needs at least two tasks. The `midi()` reads the incoming MIDI stream from the keyboard and copies it out to the next device in the chain (typically, a synthesizer). The `pdlpoll()` task polls the state of the I/O pin connected to the sustain pedal and transmits a pedal-state message when it detects a change of the pedal's state.

The `midi()` task listens to the MIDI input from the keyboard and copies whatever it sees to the MIDI output (serial output queue). More importantly, when `midi()`copies a MIDI message this way, it acquires a resource lock on the serial port queue, so it has exclusive access to this queue.

The `midi()` task in Listing 1 consists of an event loop that waits for a semaphore from the serial port interrupt handler, indicating that a character was received. A `while` loop empties the queue of received characters and feeds them into a `switch` statement, which classifies the MIDI messages by length, based on the first character and controls the display on the LCD module.

The integer variable `inseq` monitors the length of the MIDI messages. It is set to the length of the message

at the start of each MIDI message, and it is decremented for each subsequent character in that message.

When `midi()` sees the last byte of a MIDI message (signified by `inseq` reaching 0), it releases the resource lock on the serial-port queue so other tasks may use the serial port.

`pdlpoll()`, shown in Listing 2, regularly polls a pin on the 68HC11A's parallel port A. When that bit changes state, it means the pedal was pressed or released.

The `pdlpoll()` task then waits for the resource lock to be free, grabs the lock, stuffs the appropriate message into the serial port output queue (pedal up or down), and releases the lock.

If you're wondering why there needs to be a task to read MIDI in and copy it to the output, the answer lies in the fact that MIDI messages are multibyte packets. If the `pedalpoll()` task just blindly shoots out pedal-state messages, it will eventually send one in the middle of a MIDI "note on" or "note off" message from the keyboard. This will cause both messages to become garbled, and the synthesizer will throw away one (or both) of the messages as corrupt.

So, the purpose of the `midi()` task is to read and understand the MIDI stream from the keyboard. That way, it knows when it's safe for `pedalpoll()` to send a pedal state message without interrupting a keyboard message.

`midi()` uses the resource lock on the serial-port queue to let `pdlpoll()` know what times it is safe to send pedal state messages.

A third task—a device driver for the serial (MIDI) output—empties the serial port queue and sends the characters to the serial port transmit buffer register when an interrupt from the serial port transmitter indicates that the transmit buffer is empty and ready for another character. By using a semaphore, RTXC enables interrupt handlers to notify tasks of events.

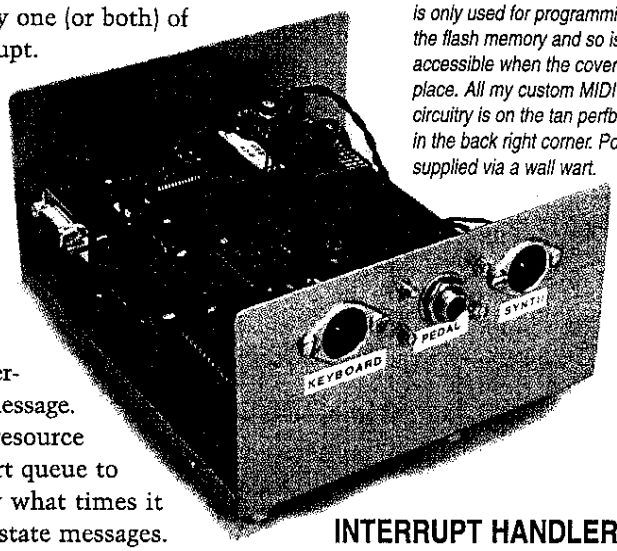

Photo 1—*The DE-9 connector is only used for programming the flash memory and so is not accessible when the cover is in place. All my custom MIDI I/O circuitry is on the tan perfboard in the back right corner. Power is supplied via a wall wart.*

INTERRUPT HANDLERS

As I mentioned, one of the interrupt handlers is devoted to the serial port interrupt. The 68HC11 has a whole slew of vectored interrupts, so it's easy to set up a different handler for every peripheral device that can generate an interrupt. When the built-in serial port of the 68HC11 receives or transmits a character, an interrupt is generated.

The handler reads the status register then responds to each of the possible interrupt sources. If a character is received, it is read and stored in a [global] buffer. And, a semaphore is set to notify `midi()` that a new character is available to be read.

If the transmit buffer-empty interrupt has fired, a different semaphore is set to notify the serial output device

Listing 2—*The `pdlpoll()` task polls the pedal switch, and when it detects a change in pedal state, waits for the MIDI output resource lock before sending the appropriate MIDI pedal message.*

```
#define SELFTASK ((TASK)0)
#include "hardware.h"

#if K4
#include <iok4.h>
#else
#include <io.h>
#endif

#include "rtxcapi.h"
#include "scidrv.h"
#include "csema.h"                  /* COMISEM, COMOSEM */
#include "cqueue.h"                 /* COMIQ,    COMOQ */
#include "cres.h"
#include "serial.h"

#define POLLED 0

static const char downmsg[] = { 0xb0, 0x40, 0x7f };
static const char upmsg[] = { 0xb0, 0x40, 0x00 };
static unsigned char ped, lastped;
unsigned char porta;

/* poll for pedal change of state, when detected, send pedal message
 * (with resource lock to prevent message scrambling). */
void pdlpoll(void)
{
unsigned char i;
unsigned char pedalsex;
#if K4
  DDRA = 0x78;              /* make it look like the port on a non-K4 */
#endif
  porta = 0x60;                     /* guess if switch is N.C. or N.O. */
  lastped = PORTA & PEDALBIT;
  pedalsex = PEDALBIT & lastped; /* look at initial state of pedal */
  if(lastped) porta &= ~PLEDBIT;
  else        porta |=  PLEDBIT;
  PORTA = porta;
  for (;;) {
    KS_delay (SELFTASK, 2);
    porta ^= PDLPOLLBIT;
    PORTA = porta;
    ped = PORTA & PEDALBIT;
#if POLLED
    monitor();
#endif
    if(lastped != ped) {
    if(ped) porta &= ~PLEDBIT;
    else    porta |=  PLEDBIT;
    lastped = ped;
    ped ^= pedalsex;                /* correct for N.C. or N.O. pedal */
      KS_lockw(SCIRES);
      porta |= SCIRESBIT;
      PORTA = porta;
      for(i = 0 ; i < 3 ; i++) {
      KS_enqueuew(OQ, (ped) ? &downmsg[i] : &upmsg[i]);
      }
      porta &= ~SCIRESBIT;
      KS_unlock(SCIRES);
      PORTA = porta;
    }
  }
}
```

driver task that the serial port hardware is ready for another character.

The second interrupt handler runs on the real-time clock interrupt of the 68HC11. This interrupt is scheduled to fire every 5 ms or so, and is used as a heartbeat by the RTXC kernel.

The kernel gets control at every real-time clock interrupt, when it checks to see if any task with a higher priority than the currently running task has become available. If so, a task switch is performed. Otherwise, the kernel returns control to the current task.

RAMPANT FEATURE-ITIS

Because a pedal is just a momentary switch, some keyboard manufacturers use normally closed switches and others use normally open switches.

My MIDI sustain pedal box would be ambidextrous. So, `pdlpoll()` checks the initial state of the pedal when the software starts up. If the pedal pin is at a high logic level, the pedal is assumed to be normally open. Otherwise, it is assumed normally closed.

This information is then used when generating the pedal state messages, to generate the correct (pedal up or down) message regardless of which pedal you plug into the device.

Besides the two five-pin DIN connectors for MIDI in and out, and the ¼ phone jack for the pedal connection, four LEDs are mounted on the front panel. These are driven by software events so I can monitor the unit's health. They started out as debugging aids, but they looked so pretty when running that I left them on the front panel of the finished device.

One LED toggles when the real-time clock interrupt fires. Another one shows the state of the pedal contact. A third shows the resource lock activity on the serial output queue. The fourth toggles at the frequency of the `pdlpoll()` polling loop.

INTERESTING COINCIDENCE

I had the project just about done when a friend showed me an article entitled "PIC MIDI Sustain Pedal" [3]. Naturally, I was intrigued.

This article solves the easier problem by generating MIDI pedal state messages. However, it assumes the keyboard and synthesizer are in one unit so they don't need to insert the pedal messages into the keyboard's MIDI output stream.

This application is simple enough for a PIC. It only has to poll the pedal contact bit and grind out serial messages—not read any MIDI stream so it can coordinate with it.

Although my MIDI box wasn't a cost-effective project to build, it certainly was entertaining and allowed me to showcase some fine tools.

Bill Dudley is a programmer for Monmouth Internet. He has designed embedded systems for the likes of AT&T Bell Labs and a whole assortment of much smaller companies. When not hacking around on computers he can sometimes be found riding one of his motorcycles. You may reach him at dud@casano.com.

REFERENCES

[1] Motorola, *M68HC11 Reference Manual M68HC11RM/A*, Rev. 3, 1991.

[2] Optrex, *LCD module*, Datasheet.

[3] R. Penfold, "PIC MIDI Sustain Pedal," *Everyday Practical Electronics*, Feb. 1999.

SOURCES

68HC11A SBC
Axiom Manufacturing
(972) 994-9676
Fax (972) 994-9170
www.axman.com

68HC11 emulator
Nohau Corp.
(408) 866-1820
Fax (408) 378-7869
www.nohau.com

68HC11 C compiler
Cosmic Software
(781) 932-2556
Fax: (781) 932-2557
www.cosmic-software.com

RTXC kernel
Embedded Systems Products, Inc.
(800) 525-4302
(281) 561-9990
Fax: (281) 561-9980
www.rtxc.com