Multi-Zone Home Audio System (Part 2)

CPU, Controls, and Development Tools

The first part of this article series introduced the multi-zone home audio system and discussed the audio hardware. This article covers the microprocessor, the controls, and the display. It also describes some ways to integrate digital audio into the system and details system issues.

By Dave Erickson (US)

I used a microprocessor to control analog signal processing in my multi-zone home audio project. This approach is a good match for my capabilities and meets all of my goals. After all, analog is my favorite programming language.

A system could be set up digitally. In that case, the inputs would be either digital audio devices or, if they are analog, they would use an ADC for input. Then the source selection, volume, tone (filters), and other functions would be done digitally, either in firmware or by a DSP or FPGA. You could then send the sound digitally to the zones where a DAC or a "digital amplifier" could be used. Or why not distribute all the inputs to all the zones and make each zone a digital audio system?

DIGITAL VS. ANALOG AUDIO

A digital system would be outside my current skill set and would exceed my cost and complexity (design effort) budget. Ethernet is appealing for a wired system. Several pro-audio Audio over Ethernet (AoE) systems do this, but they are mostly proprietary and do not support Wi-Fi. Sonos is one commercial and proprietary Wi-Fi-based system designed for homes.

One AoE challenge is managing latency delays. Ethernet is not real time, so latency and thus delays are not controlled. Room-to-room delays of more than a few milliseconds would create objectionable echoes. I am not aware of any open solutions that offer delay matching.

One advantage of a centralized analog system is that the electronics can be located in one compact box. With analog, either speaker wires or line-level audio are run to each room. Wiring is simpler in a single-story home, but if you are an enterprising engineer who lives in a multi-story home, running a few wires probably won't stop you. With electrical, phone, cable, networking, sensors, audio, and so forth, I have a lot of wires in my house. It is simple to add a few more.

The multi-zone home audio system's cost per channel is roughly $120 to $150 per zone in an eight-zone system, including amplifiers and speakers. A zone that already has powered speakers, a PC, or a boombox costs about $60 since line-level audio can be wired to existing powered speakers. This cost does not include the labor of stuffing boards, loading code, and running wires.

Speakers can be just about anything including in-wall, in-ceiling, high-end, bookshelf, or indoor/outdoor types. I use a separate high-power amplifier and home-built speakers for my living room. **Figure 1** shows the system's design.

CPU AND CONTROLS

A home audio system needs several different but similarly functioning controls such as IR remote control to satisfy couch potatoes; front-panel controls for when the control is misplaced, or to do more complicated setups; remote zone controls to adjust the volume, source, or sound without having to return to the main system; and a display to view the status. Also, why not include a digital interface to enable system control from a PC? The challenge of all these controls is to make them act in a similar manner despite their very different electrical interfaces.

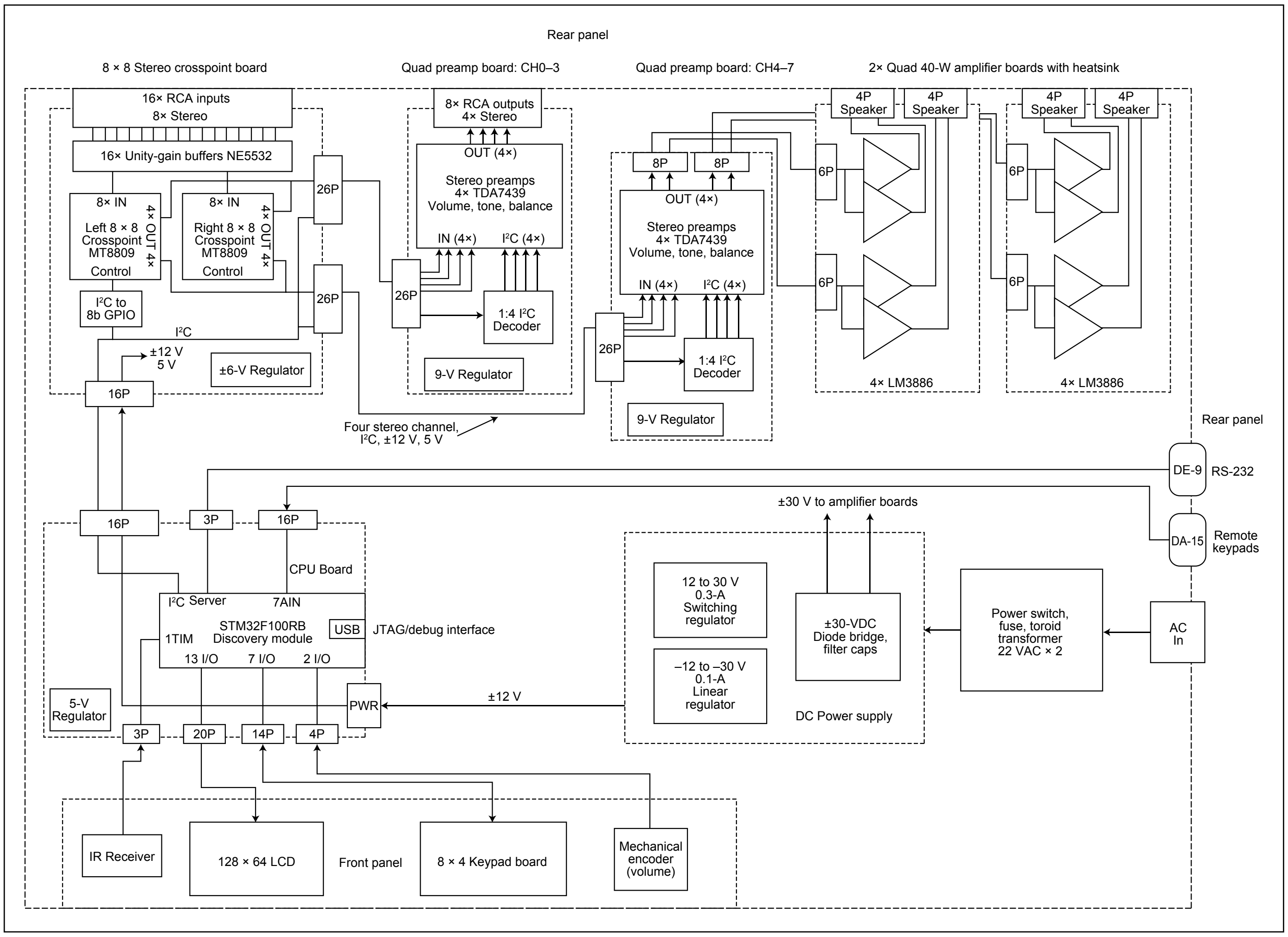

Rear panel
8 × 8 Stereo crosspoint board
Quad preamp board: CH0–3
Quad preamp board: CH4–7
2× Quad 40-W amplifier boards with heatsink
16× RCA inputs 8× Stereo
16× Unity-gain buffers NE5532
8× IN Left 8 × 8 Crosspoint MT8809 Control 4× OUT 4×
8× IN Right 8 × 8 Crosspoint MT8809 Control 4× OUT 4×
I²C to 8b GPIO
I²C
±12 V 5 V
±6-V Regulator
16P
26P
26P
8× RCA outputs 4× Stereo
OUT (4×) Stereo preamps 4× TDA7439 Volume, tone, balance IN (4×) I²C (4×)
1:4 I²C Decoder
9-V Regulator
8P
8P
26P
4P Speaker
6P
4× LM3886
Four stereo channel, I²C, ±12 V, 5 V
Rear panel
DE-9 RS-232
DA-15 Remote keypads
AC In
±30 V to amplifier boards
16P
3P
16P
CPU Board
I²C Server 7AIN
STM32F100RB Discovery module
1TIM
USB JTAG/debug interface
13 I/O 7 I/O 2 I/O
PWR
±12 V
5-V Regulator
3P 20P 14P 4P
12 to 30 V 0.3-A Switching regulator
–12 to –30 V 0.1-A Linear regulator
±30-VDC Diode bridge, filter caps
DC Power supply
Power switch, fuse, toroid transformer 22 VAC × 2
IR Receiver
128 × 64 LCD
Front panel
8 × 4 Keypad board
Mechanical encoder (volume)

FIGURE 1
The system block diagram includes the boards, controls, amplifiers, and power supplies. The approximate component positions in the chassis are also shown.

GUI AND CONTROL STRATEGY

There are several ways to implement soft controls and displays. At one extreme, a multi-level menu system with soft keys can maximize the functionality of a few simple controls. On the other end of the spectrum, having one control per function enables you to perform functions without navigating menus.

Think of a calculator or an analog stereo with knobs or buttons for each basic function. Since it is important that my non-engineer family members can easily use the system, I used a graphic LCD combined with many labeled buttons for feedback.

Eight buttons select the eight zones, and eight more select the sources. Buttons for volume up and down, balance, bass, midrange, and treble are also included. There is a mute button as well as a mute-all button.

There aren't any multi-level menus yet. For example, to send the PC sound to the kitchen and adjust its volume, first select the zone, kit, then the source, PC, and then use the Volume Up and Volume Down buttons to adjust the volume.

Changes are displayed on the LCD in real time. This interface is fairly simple and intuitive. The buttons are arranged in a four-row by eight-column matrix. I color coded the buttons to help distinguish the functions. For control labeling, I used three 9-mm label tapes: one for the zones, one for the sources, and one for the other controls. I used arrays of ASCII strings for software labeling of the zones and the LCD's inputs. This required a recompile to change the house or input configuration.

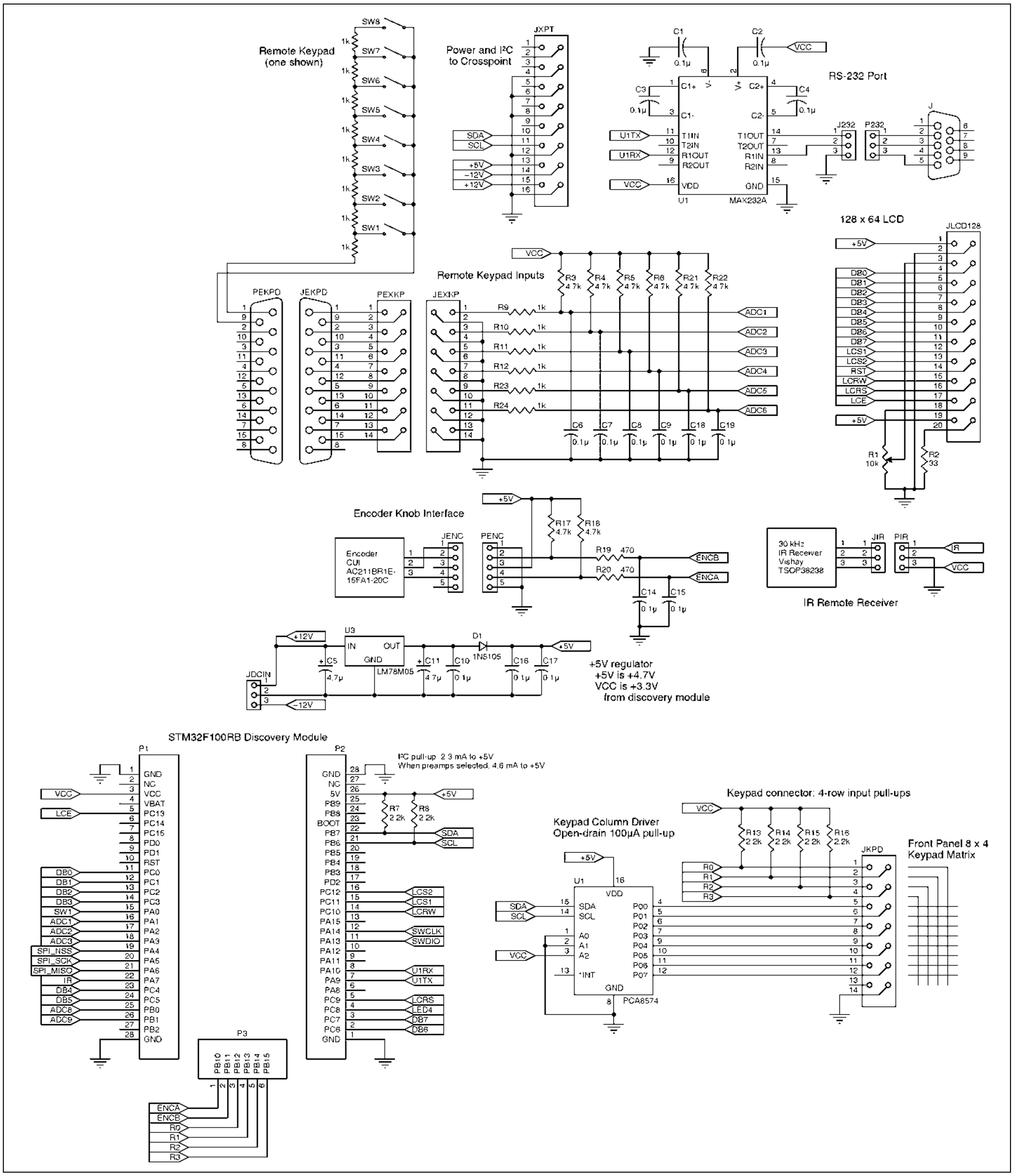
Basic functions are repeated on the IR remote control and the remote keypads. Since these functions do not have visual feedback, controlling a remote zone could create problems as you continuously increase the volume but hear no response. Meanwhile, the neighbors four houses away can hear your deck speakers just fine.

So, to avoid trouble, I currently control only the local zone with the remote keypads. If you need to control a different zone, you need to get off the couch and go to the front panel. It would be amazing if the system had a smartphone app or at least a web browser interface.

CPU BOARD

Figure 2 shows the CPU board schematic. It contains an STMicroelectronics STM32VLDISCOVERY board as a daughtercard and interfaces for the LCD, analog remote keypads, the eight-row by four-column front-panel keypad, IR, the encoder knob, RS-232, and the I²C connections for the preamplifier control. The CPU board is built as a hand-wired prototype but I plan to layout a PCB. **Photo 1** shows the front-panel setup.

FIGURE 2
This is the CPU board schematic.


Remote Keypad (one shown)
Power and I²C to Crosspoint
JXPT
RS-232 Port
MAX232A
128 x 64 LCD
JLCD128
Remote Keypad Inputs
PEKPD
JEKPD
PEXKP
JEXKP
Encoder Knob Interface
Encoder CUI AC2118R1E-15FA1-20C
JENC
PENC
30 kHz IR Receiver Vishay TSOP38238
IR Remote Receiver
LM78M05
1N5105
+5V regulator +5V is +4.7V VCC is +3.3V from discovery module
JDCIN
STM32F100RB Discovery Module
I²C pull-up 2.3 mA to +5V When preamps selected, 4.6 mA to +5V
Keypad connector: 4-row input pull-ups
Keypad Column Driver Open-drain 100µA pull-up
PCA8574
JKPD
Front Panel 8 x 4 Keypad Matrix

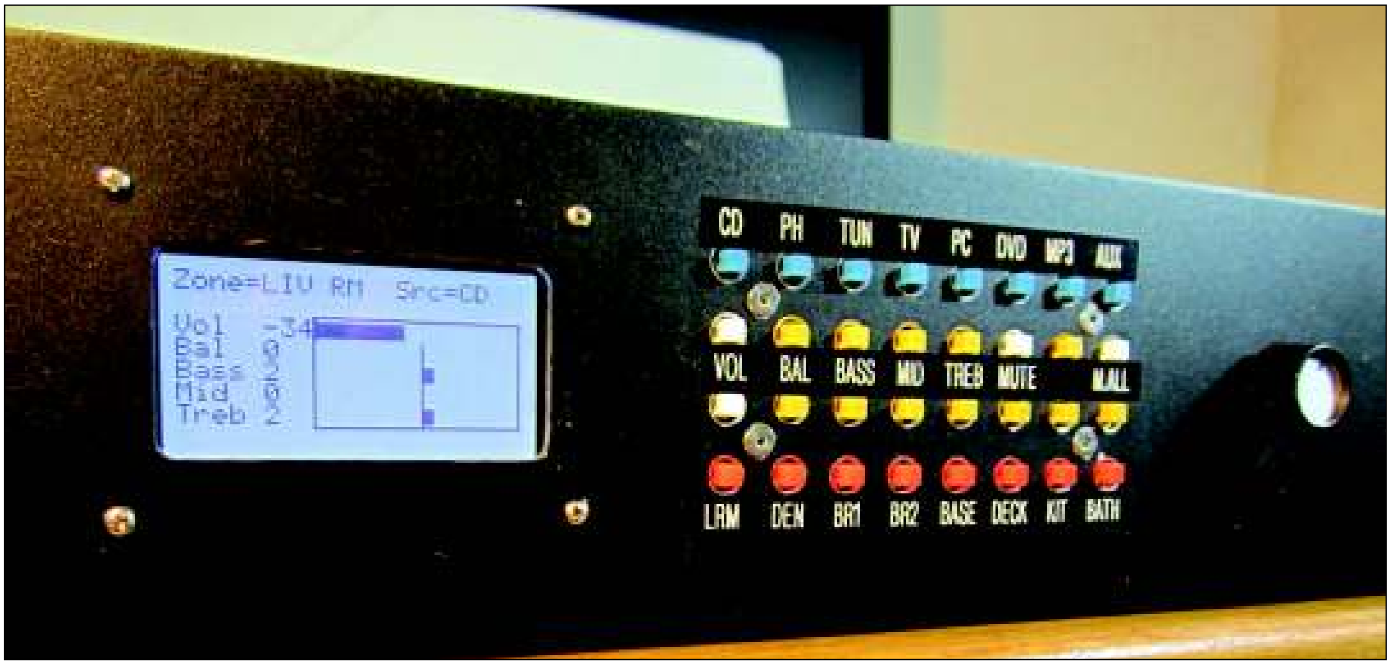


PHOTO 1
The front panel includes an LCD, keypad, and encoder knob.

FRONT-PANEL KEYPAD

The front-panel keypad is an ExpressPCB board containing 32 6-mm momentary buttons, organized as eight rows by four columns. The eight rows are driven by a PCA8571 I^{2}C GPIO port. Its outputs are open collector and pulled high, so pressing multiple keys causes no damage. The four columns are wired to processor input pins. This keypad is scanned one column at a time in a 1-ms timed-interrupt routine.

IR REMOTE

I used Sony remote control codes and emulated a Sony receiver. Sony codes use a simple protocol: a Start bit followed by 12 data bits. A Start bit is 2.4 µs wide, a Zero bit is 0.6 ms, and a One bit is 1.2 µs. The five most-significant bits (MSBs) of the 12 bits are the device code (CD, TV, DVD, etc.) and the seven least-significant bits (LSBs) are key codes.

The IR receivers detect the IR light, pass only the 38-kHz carrier frequency, and demodulate it to a CMOS logic signal. The firmware decodes these pulse widths and assembles the code.

If you haven't dealt with an IR decoder, it is all about error detection. Basically if you ever detect an error, toss out the code. By this, I mean reset a bit counter and then wait for the next Start bit.

The STM32VLDISCOVERY board's timers can directly measure pulse widths applied to input pins and generate an interrupt that informs you when a pulse has arrived. The next step is to read a register to get the pulse width and then compare the pulse widths to the minimum and maximum values for each of the Start, Zero, or one-pulse widths, which requires a maximum of six comparisons.

I clocked the 16-bit timer at 1 µs to provide plenty of resolution and to make the math convenient. A state machine waits in State 0 for a valid Start pulse to arrive. When it does, it counts the number of valid Zero or One bits and shifts them into a 16-bit value. If pretty much anything goes wrong, reset the state machine to a zero count. By "going wrong," I mean detecting any pulse that is not a valid one or zero for the next 12 pulses. When the count hits 13 (Start bit plus 12 data bits), you have a valid IR code. Well, almost.

For another test, count the number of identical IR codes in a row and only generate a valid key code when two identical codes arrive. IR remotes are intended to work this way, generating at least three identical codes in a row.

If you hold a button down, it continues to send the same code. Imagine a code being optically interrupted and then the end of another code arrives. Requiring two identical codes prevents this type of error.

Why do all this testing? We are surrounded by IR noise (e.g., from sunlight, incandescent and other lights, different remote controls, etc.). Also the IR signal can be weak, interrupted by the cat, and so forth. IR receivers contain a 38-kHz band-pass filter to help eliminate most ambient light, but this does not remove pulses from other IR remotes. All this may sound complicated, but is only about 20 lines of C and can be done in the timer interrupt handler. I have reliably used this method on this system and the original Freescale Semiconductor (formerly

LISTING 1
An array of these control data structures is used, one per control.

```
/* Controls structure: one per keypad. */
typedef struct {
  uint8_t  rawCode;      /* Key before LUT */
  uint8_t  rawFlag;      /* Key down from scan */
  uint8_t  keyCode;      /* Key after LUT */
  uint8_t  state;        /* State: 0-debounce 1-repeat dly 2-repeat*/
  uint8_t  sendFlag;     /* It's good. send it */
  uint16_t repeatTimer;  /* 1ms timer for key auto-repeat */
  uint8_t  repeat;       /* 1 if key should auto-repeat */
  uint8_t  zone;         /* Current zone for this keypad */
  uint16_t zoneTimer;    /* Time until this zone resets to default */
  uint8_t  zoneDefault;  /* Zone to default back to */
  uint8_t  oldKey;       /* Previous key for filter */
}inKeyTypeDef;
```

Motorola) 68HC11 microcontroller.

For a remote control, I used a One-For-All eight-channel universal remote to control all the equipment in my living room. These are low cost and readily available.

REMOTE KEYPADS

For each remote zone, an optional keypad provides basic controls including volume, source selection, balance, and muting. Remote keypads should be simple, inexpensive, and reliable. The microprocessor's eight-channel 8-bit ADC was unused on the original system.

An ADC is a terrible thing to waste, so I used it to read the remote keypads. Each keypad has eight momentary switches and a resistor ladder consisting of seven 1-kΩ resistors in series (press Button 0 and you get 0 kΩ, press Button 1 and you get 1 kΩ, etc.). In this way, up to eight remote keypads can be read, using one ADC channel per keypad.

Back on the CPU, a simple passive circuit is needed to read each channel's resistance. Each ADC input has a 4.7-kΩ pull-up resistor, a series 1-kΩ resistor, and a 0.1-µF filter capacitor. The 1 kΩ in series protects the analog pin in case of a static zap. Ideally, there should also be a protection diode on each input.

To detect a key press, the firmware periodically reads the ADCs, checks the voltage to see if any key is pressed, compares the voltage to the high and low thresholds for each of the eight keys, and increments a counter to debounce the keys.

I measured only one channel in the 1-ms timed interrupt routine. In fact, I triggered the ADC to measure the next channel after reading the current channel. This way, the interrupt routine doesn't need to wait for the ADC. Then the `key_proc()` code looks for 50 ms of the same keys detected to confirm a valid key press.

Keypad wiring uses a single unshielded twisted pair from the keypad back to the system. Polarity does not matter. However, running wires is inconvenient, so I am looking for a web interface to enable a PC or smartphone to control the system through a simple app or webpage. More on this follows.

ENCODER KNOB

Mechanical encoder knobs are inexpensive and provide a nice feel. Since volume control is important to this system, I decided to add a real volume knob. Like most encoders, the knob uses a two-bit quadrature code. There are many types of encoder signal timings. Some encoders output four 90° steps per mechanical click. The one I use outputs one 90° step per click.

The 1-ms interrupt routine reads the two encoder bits and combines them with the previous reading to build a simple 4-bit code from the two readings. Since this code contains both the current and previous states of the knob, a simple 4-bit (16-entry) look-up table (LUT) can determine what action to take.

The LUT has entries for Up, Down, or NOP. Using a LUT enables the code to be changed to accommodate various manufacturers' mechanical configurations. This LUT value determines whether a count value is incremented, decremented, or unaffected. The changes in the count can then be read in the main routine to change a setting (volume for now) based on whether the counter has increased or decreased.

MAKING CONTROLS CONFORM

Detecting a key press is simple. Debouncing and making the key periodically repeat after being held a while is a bit more complicated. Making 10 or so sets of controls comprising three totally different electrical types (front panel buttons, remote analog keypads, and an IR remote control) all act the same requires some thought.

Key auto repeat is useful for volume and other controls. To auto repeat, time how long the same key is pressed and, after a few hundred milliseconds, send another key code. Then wait a few hundred more milliseconds and resend it.

The IR remote presents a problem. Each remote generates codes at its own rate and repeats when the key is held down, but not necessarily at the repeat rate that you want. So a filter is needed that turns on when a key is pressed and times out after a period of time when the key is released. Then the IR buttons can be treated the same as any keypad.

Another issue is that any zone can be controlled by the front panel. But mostly you want to control the local zone from that zone. So, after a timeout period of no activity, each panel defaults back to the local zone.

I do not currently control any other zone from the remote keypads. When I do add this feature, these will have the same timeout mechanism.

If you combine all these requirements, the code can become complex. Fortunately, data structures are your friends. In **Listing 1** the struct `TypeDef` is used to control each keypad. An array of these structs is used, one for each keypad.

In addition to the multiple control sources, during system debugging it is beneficial to use a PC keyboard to control the system and a PC display to output debug `printf()` messages via a terminal emulator and RS-232. I used a single keyboard ASCII key to emulate each control.

For example, "V" is volume up and "v" is volume down. The right inputs are selected by "1" through "8." The eight zones are selected by "SHIFT_1" through "SHIFT_8." It is helpful to choose commands you can remember. I use "h" for Help to display a list of the commands.

A big `case` statement interprets and executes all the commands and uses these single ASCII codes as its selector. The other

```
Set X address (x & 0x3F)
     Set bus direction OUT
     Set RW, RS, both CS
     Set data
     Pulse E
E= 1, Delay 500ns, E = 0
     Delay(5us)
Set Y page (y >> 3)
     Set data
     Pulse E
     Delay(3us)
Pixel mask (1 >> y%7)
Read the dummy data
     Set bus direction IN
     Set RW, RS
     Select L or R chip based on X>63
     Pulse E
     Delay(5us)
Read the real data
     E = 1
     Delay(1us)
     Input data
     E = 0
     Delay(3us)
Reset X address (x & 0x3F) since the last read incremented it
     Set bus direction OUT
     Set RW, RS, both CS
     Set data
     Pulse E
     Delay(3us)
Merge the mask and the read data:  (mask | data)
Write data
     Set bus direction OUT
     Set RW, RS
     Select L or R chip based on X>63
     Pulse E
     Delay(3us)
```

LISTING 2
This is the pseudocode for `putpix()` to write a single pixel to the display.

various controls use a small LUT to map their binary outputs into the same ASCII codes. I invented this simple method long ago and suspect that many others have also come up with it.

A nice side effect of this technique is that your project has the foundation of a serial protocol to control it remotely. If you add a serial to USB chip, then presto! Your project has a USB interface.

GRAPHIC LCD: THE UBIQUITOUS KS0108

The goals for a front-panel display are to show all of the parameters of one zone at a time and to graphically and interactively show the volume, balance, and tone settings. The LCD should be small, but not too small; highly visible in any lighting; and low cost. I like the look of white LED backlights. I decided to use a 128 × 64 monochrome panel based on the KS0108 controller chip. These low-cost chips are readily available from several manufacturers. I chose the NHD-12864WG-BTFH-V from Digi-Key, which costs about $20.

"A home audio system needs several different but similarly functioning controls. The challenge of all these controls is to make them act in a similar manner despite their very different electrical interfaces."

The hardware interface is straightforward. It has eight data bits and five control signals. The KS0108A uses two devices; each accesses one half of the display, so two chip selects are needed.

One of the first tasks of this project was to write the LCD hardware access-level code. I found a few examples online for KS0108 code and borrowed a few ideas, but finally decided to write my own.

I have a higher-level LCD library, which I have used with my own FPGA-LCD based controller designs since the 1990s (see my article "Graphics LCD Control for Embedded Applications," *Circuit Cellar* 34, 1993). Using an off-the-shelf panel with its built-in controller was a real experience, since these chips have a few quirks. They are byte-oriented and the pixels within a byte are vertically organized. Since most font and bitmap files are organized with horizontal bytes, they would need to be transposed either by the microprocessor or before they are loaded into code. Fortunately, I found a 5 × 8 font designed for the KS108A. I haven't needed bitmaps yet, so I haven't had to face that challenge.

GRAPHICS FUNCTIONS

To make good use of a graphics LCD, you need functions to draw common objects. Lines, filled rectangles, and ellipses (circles) require a pixel draw routine. Characters and bitmaps need a byte write function.

Display clearing and updating should be fast enough that you don't notice flicker between the time that you clear the display and the time that you write the new data. It should be just a few milliseconds to avoid flicker. The fast 24-MHz ARM Cortex-M3 processor can do the job without a lot of special optimizations. I used a 1-MIPS 8-bit 68HC11 microcontroller and a larger LCD in this project's previous version, so the code had to jump through hoops to update the LCD fast enough to prevent flicker. The FPGA-based LCD controller helped since it was designed to make drawing primitives fast enough with even a slow processor.

The KS0108 vertical-byte data organization requires that fonts are eight pixels high including spaces. Otherwise, you would need to write them one pixel at a time, which would be quite slow. So the only practical small font is 5 × 7. Having your characters placed anywhere vertically except on a byte-boundary is pretty painful so I accept this limitation.

Larger fonts fit into two or more bytes, but I don't currently use these. One approach I have used to generate larger characters is to pixel replicate the small font by two or three times. This has the advantage of using only one small font table but the disadvantage that larger fonts can appear blocky.

I like to render lines and circles in the LCD memory, so I needed to write graphics one pixel at a time. Sounds simple, right? Unfortunately with the KS0108 such a seemingly simple operation requires an inordinate amount of code. **Listing 2** shows the pseudocode for a `putpix(x, y)` function just to write one pixel.

Part of the complexity is the large number of delays needed to access this older slow device and to meet all its setup and hold timings. A pixel write requires you to do a read/modify/write to the memory and single byte reads are inefficient. To do a read, first a dummy read is required. Then, since the X address always auto increments, the address needs to be set back again before the real read. I used a 24-MHz processor with plenty of code space, but the delays alone add up to about 25 µs. I estimate about 40-µs total CPU time per pixel. I have not measured it.

One thing that the KS0108 does reasonably well is to move a block of data from CPU memory to the LCD. That is because the addresses increment automatically after an access. So writing byte-aligned fonts is fairly fast. Another approach to manage a display is to render the entire display in CPU memory much more quickly and then move the entire screen to the LCD (128 × 64/8 is only 1 KB). But for larger panels, this approach uses a lot more CPU memory and more time to update the screen. For example, a 320 × 240 1-bit panel needs 9,600 memory bytes, so it would require about 10 times as long to move that data. Rendering graphics in the display minimizes CPU memory use.

Currently only two screens are displayed: a startup "splash" screen that displays the code revision and date and a single-zone status screen. I plan to add at least one more screen, using large characters, so I can read the source and zone from across a room.

DEVELOPMENT TOOLS

The STM32VLDISCOVERY board modules are a great deal. They offer a DIP module that brings out every pin of the 64-pin processor and the ST-LINK USB debug interface, which is STMicroelectronics's two-wire JTAG programming and debug interface, all for $12.

STMicroelectronics's 2011 STM32 Design Challenge offered a free starter version fAtollic's C development tools. Unfortunately, when the contest ended, Atollic imposed a 32-KB code size limit in its free version. Since my code was already 49 KB and growing, this was a problem.

After looking at Yagarto and other toolsets, I found CooCox's CoIDE, an integration of Eclipse, GCC, GDB, a lot of programmer and debugger support, and nearly every ARM CPU manufacturer's device libraries, all free and without limitations. CooCox's website has many user-supplied examples. Porting to CooCox was fairly painless, as is changing to other ARM devices.

ARM PERIPHERALS

I am used to the peripherals on 8-bit devices, most recently the Atmel AVR microcontroller. I was surprised at the extensive features and complexity of STMicroelectronics's ARM Cortex-M3 devices.

For example, all the GPIO ports are 16 bits and each bit can be controlled several ways by multiple registers. In addition to writing all 16 output bits or their direction register, there are multiple-bit Set and Clear registers for both the data and the direction registers. These enable multiple device handlers to access the same port's individual bits without interference. On typical 8-bit processors, care must be taken to prevent interference between multiple device handlers.

There are several of each type of peripherals. On this mid-end processor, there are multiple ADCs, DACs, I^{2}Cs, UARTs, SPIs, timers, and so forth. Using an I/O register's name directly is not a practical way to handle multiple devices since too many unique names would be required.

STMicroelectronics offers extensive libraries to help write device code. At first I was intimidated by the dozens of functions just to access an I^{2}C device, for example. However, once you figure out what functions you need to do your job, the rest goes smoothly.

Even device initialization can be daunting when there are dozens of registers with possibly hundreds of bits. To help, ARM uses a configuration data structure and provides an initialization function such as `DAC_Configuration(void)`. First you set the structure elements, then you call the function to transfer the struct to the device. All devices can be initialized in this way.

DIGITAL AUDIO SOURCES

In our home, we currently use a standard desktop PC in the den as a music server to play our MP3 collection via Winamp and to stream audio from Pandora or other services. For a local MP3 player, a simple mini-plug to RCA cable will do the job, but it does not

PROJECT FILES

circuitcellar.com/ccmaterials

RESOURCES

Atollic Inc., www.atollic.com.

D. Erickson, "Graphics LCD Control for Embedded Applications" *Circuit Cellar*, 34, 1993.

Digi-Key Corp., www.digikey.com.

ExpressPCB, www.expresspcb.com

Raspberry Pi, www.raspberrypi.org

Sonos, Inc., Sonos system of hi-fi wireless speakers and audio components, www.sonos.com

Yagarto, Yet Another GNU ARM Toolchain, www.yagarto.org

SOURCES

ARM Cortex-M3 processor
ARM, Ltd. | www.arm.com

AVR Microcontroller
Atmel Corp. | www.atmel.com

BeagleBone Black
BeagleBoard.org | www.beagleboard.org.

CoIDE Software development environment
CooCox | www.coocox.org

68HC11 Microcontroller
Freescale Semiconductor, Inc. (formerly Motorola) | www.freescale.com

STM32VLDISCOVERY board and ST-LINK in-circuit debugger/programmer
STMicroelectronics | www.st.com

ABOUT THE AUTHOR

Dave Erickson (dave@djerickson.com) has been an electronics hobbyist since the 1960s. He earned his BSEE in 1976. Dave worked at HP Medical, Datacube, Analogic, Zoll Medical, Teradyne, and numerous startups. He currently develops electro-optic and ultrasound systems for a cardiac catheter system at Infraredx. Dave's electronics interests include instrumentation, audio, electronic music, and boat electronics. He also enjoys biking and sailing. His projects are available at www.djerickson.com.

provide either charging or a stand. iPod docking stations with audio outputs can be purchased for that purpose.

GROUNDING AND HUM

Most consumer audio gear is not power line grounded and uses two-wire line cords. The various RCA audio connections provide a local "ground reference" and things work well enough. Connecting to a single grounded audio source doesn't generally create a ground loop or hum problem. Connecting two or more grounded sources creates a ground loop and, depending on the power line ground-voltage difference, can causes varying amounts of hum.

I chose to ground my system since it is the center for many audio signals throughout our house. Connecting to ungrounded equipment doesn't present a problem, but connecting to other grounded equipment can cause hum.

Examples of grounded audio components are PCs and most cable TV boxes. A cable box's RF cable is grounded where it enters the house for lightning protection. If you connect your PC or cable box to an audio system and don't get hum, great. If there is hum, commercial audio isolation transformers will solve the problem.

FUTURE FEATURES

One feature on my wish list is the ability to control the system from a web page. This would enable a smartphone or PC anywhere in the house to control the system. One reason I used a powerful ARM processor when a lesser CPU would probably do was to have the resources to someday serve up webpages and handle TCP/IP.

You may ask, "So where is the webpage, Dave?" The truth is that I do not currently possess the skills to generate active webpages and handle file systems, TCP/IP stacks, and so forth.

I have seen projects that implement simple web servers on an 8-bit processor and I consider these interesting, but that is all. They typically do not have a real server, a file system (for images, HTML, etc.), or file management tools. They use a lot of `sprintf()` commands to render HTML or JavaScript on the fly. They typically do not have DHCP or system configuration tools.

If you open a port to the Internet, you will need to deal with security issues. After using high-level tools and real servers to write web pages, this approach seemed primitive.

I like to spend my hobby time developing skills and systems that are applicable to my career or at least to a real commercial product. I tend to avoid developing toys, tricks, and hack code. So as I wait patiently for someone to drop a nice web server, file system, and TCP/IP stack for STM32 processors in my lap, time marches by.

At this point, I am leaning toward using a low-cost, low-power Linux board (e.g., a Raspberry Pi or BeagleBoard.org's BeagleBone Black) to address these features. They support Apache and other web servers with Internet security, full local and network file systems, music (and video), and servers (e.g., XBMC), all written and supported by serious programmers. And it all comes in a credit-card size $50 board that consumes a watt or so.

A web browser, Pandora, or any other streaming web function is just a download away. I have not worked much with Linux on embedded controllers, but it is a skill I would like to develop. So many projects, so little time.

BUILDING YOUR OWN

I used ExpressPCB to design the boards. If there is interest, I will offer bare PC boards on my website. The boards are designed with surface-mount technology (SMT) electronics, except for the connectors and the film capacitors, which are through-hole. The SMT parts are mostly 0.05" pitch and 0805 or larger and can be built under a magnifier by hand-soldering. I used ribbon cables where possible to minimize cable assembly labor. The CPU board is currently hand-wired. The ExpressPCB layout in available on the project website.

This has been a rewarding project for me. If you are interested, the full ExpressPCB schematics, PCB artworks, BOMs, and code are available on *Circuit Cellar*'s FTP site. The project website is available at www.djerickson.com/multizone.