Well, the fireworks state machine \texttt{FWState()} would be a good place to test for that condition. The main program could still schedule the launch, but the \texttt{FWState()} would scrub the launch if winds were too high. And adding this requirement doesn’t complicate the design.

This, of course, was a very trivial example of good state machine design and how you can be lead into a too complicated design. And we also discussed recovery from all that complication by dividing the design into separate state machines.

\textbf{RTOS}

In the future I’ll cover the topic of completing this type of design with an RTOS. The state machines that we’ve been talking about begin to look like tasks in an RTOS. So, if you’re not currently using an RTOS and would like to (or need to), just work on getting your state machines working well and you’ll be ready for that next step.

George Martin (gmm50@att.net) began his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and co-founded a design and manufacturing firm (www.embedded-designer.com). His designs typically include servo-motion control, graphical input and output, data acquisition, and remote control systems. George is a charter member of the CircAl Design Works Team. He is currently working on a mobile communications system that announces highway info. He is also a nationally ranked revolver shooter.

\textbf{Need-to-Know Info}

\textbf{Knowledge is power.} In the computer applications industry, informed engineers and programmers don’t just survive, they \textit{thrive} and \textit{excel}. For more need-to-know information about topics covered in George Martin’s Issue 242 article, the Circuit Cellar editorial staff highly recommends the following content:

\textbf{Embedded Breakup}

\textit{Divide a Design and Minimize Processing}

by George Martin

\textit{Circuit Cellar 230, 2009}

Once you have your embedded processor up and running, what’s next? George presents tips for partitioning design work and minimizing processing requirements. Topics: State Machine, Coding, Processing, Partitioning

Go to: www.circuitcellar.com/magazine/230.html

\textbf{Hardware Synthesis with VHDL (Part 2)}

by Michael Griebling

\textit{Circuit Cellar 182, 2005}

Ready to put VHDL in play? Here you learn to build state machines, decoders, and more. Topics: State Machine, Buffers, Registers, Decoder, Multiplexer

Go to: www.circuitcellar.com/magazine/182toc.htm

\textbf{Got Serial, Need Network?}

\textbf{AP CIRCUITS}

PCB Fabrication Since 1984

As low as... $9.95 each!

Two Boards
Two Layers
Two Masks
One Legend

Unmasked boards ship next day!

www.apcircuits.com

\textbf{gridconnect}

www.gridconnect.com

+1 800 975-4743

\textbf{September 2010 - Issue 242}
Transmit and Decode Data
Design and Implement a Keyfob Decoder

Data transmission technology has come a long way during the past few decades. Today, wireless technology is everywhere. In this article you learn how to design and implement a keyfob decoder.

In my youth, I had an appreciation for Lionel trains and Erector sets. I distinctly remember one of the projects described in the Erector set booklet incorporated electrical construction as well as mechanical. A couple of batteries were connected to the AC motor’s plug, along with a couple of hand-held metal handles. In series with the batteries, I added a switch consisting of a gear turned by a crank. An insulated metal girder was positioned so that its end made and broke contact with the gear’s teeth as it was rotated. This provided a repeated switch like toggling to the current running into the motor and the parallel handles.

So, I would have one of my sisters grab the handles while I cranked the little gear, providing her with an electronic “tickle” as the motor’s field coils alternately energized and collapsed. To this day, I can’t believe that this electrocution engine was a documented project. My three younger sisters were gluttons for the punishment I doled out. I feel like I need to apologize to them for this. Sorry, girls!

Those episodes were probably my first encounter with electricity. When I received my first Lionel train set, I became entranced with remote control. Even though this was “wired” remote control via the tracks, I could start and stop the locomotive from afar and even change its route via track switches. Today, train enthusiasts have small CCD cameras mounted in the engines that give the driver an engineer’s point of view of the action. I had to use my imagination back then!

No longer do we consider “wired” to be remote control. All-out remotes are now wireless. They might be IR, RF, or even audio. Remember The Clapper sound-activated on/off switch? Many of you carry a remote in your pocket to authorize access to your car. When I wanted to give one of my projects an optional manual control of its X-Y positioner, I opted for a wireless link. As this project was mounted beyond my reach, I wanted to be able to manually adjust it without having to set up an extension ladder each time.

BOM
After putting together the bill of materials for a key chain RF dongle, I scratched my head and thought: “There must be an easier and cheaper way to get this done.” While thumbing through a parts catalog, I came across the answer I was looking for. Linx Technologies makes a line of transmitter keyfobs. At less than $20, it looked like the way to go. I chose a CMD-KEY5-433 keyfob, which has five push buttons and transmits an on-off keying [OOK] signal on 433 MHz. The keyfob employs an LR series transmitter with a Holtek HT640 encoder IC. While these come in 315-, 418-, and 433-MHz versions, only the 433-MHz version is precertified for FCC part 15.

PROTOCOL
For those of you unfamiliar with Holtek encoder/decoders, here’s a little background. The Holtek transmission is not the simple 8-bit
data format you might find in a typical serial connection. It was developed for security—that is, it wants to provide information, but only to a specific receiver, like in your car, and not all of the vehicles within range. It does this and keeps the transmission durations short by sending a message totaling only 20 bits. There is a 8-bit sync period (6 bits of silence followed by 2 sync bits), a 10-bit address of the receiver it is paired with, and 8 bits of data. Note: except for the (minimum) 6-bits of silence, every bit begins in the high (1) state. Unlike a UART’s data, there are no start and stop bits for each byte of data. It’s more like an SPI transmission without the clock line.

Only those receivers that have been set to the corresponding (10-bit) address will offer the data received to the user. Non-matching receivers just dump the data. So, normally, you will use a matching Holtek decoder after the RF receiver to automatically handle the transmitted secure format.

What happens when you already have access to a microcontroller in your circuitry? Can’t you eliminate the middleman (Holtek decoder) and just decode with the microcontroller? Yes, you can. By understanding the Holtek format, you can use the microcontroller to receive the encoded data, decode it, and take any action necessary for our application.

If you refer to Figure 1, you’ll notice the familiar OSC inputs. These pins are not crystal connections but a user-selectable “R” for an internal RC oscillator running from 20 to 200 kHz (330 kΩ 5% ~ 100 kHz). Also, note that encoder has 8-bit data and 10-bit address inputs. A data clock is derived by dividing the internal oscillator by 33. One bit equals six clock cycles (see Figure 2). Each bit consists of one of two patterns of three cycles each. A pattern can be LLH (low for two data clocks followed by a high for one data clock) or LHH (a low for one data clock followed by a high for two data clocks).

You might have noticed three things when you looked at the bit pattern interpretation. Let’s review.

One, a bit is considered a “1” if the pattern is the same LLH and LHH. A bit is considered a “0” if the patterns are the same LHH and LHH. Or, a bit is considered an “open” if the patterns are different, an LLH followed by an LHH. Each bit is therefore a trinary value and holds 50% more data than a binary value. Data and address inputs have three states: pulled up [1], pulled down [0], and left floating [open]. In reality, since a bit consists of two patterns, it can be argued that each bit is actually 2 bits and thus holds 25% less data than a 2-bit binary value.

Two, there is an undefined pattern in Figure 2. The pattern LHH followed by an LLH has not been defined. Actually, this pattern is the sync pattern used to indicate the start of a new transmission. More on this shortly.

And lastly, each pattern begins with a LOW state and ends with a HIGH state. This means every pattern must begin with a HIGH-to-LOW transition. This is important for getting synchronized with a transmission burst.

Figure 3 shows the data structure of a transmission. You’ll notice it begins with a period of silence, a minimum of 6 bits in length. The first data to be transmitted is a 2-bit sync pattern.

This is used to synchronize the receiver. This is the only pattern a decoder will (and should) recognize as legal. A decoder must disregard the total transmission unless it has recognized this preamble.

The actual address and data bits follow in an LSB first format. Once a button press initiates a transmission, it will complete even if you release the button. Transmissions will repeat (including the 6-bit silence period) as long as the button is held down.

RF
This keyfob contains a transmission carrier in addition to the Holtek encoder. The Linx Technologies TXM-433-LR transmitter module (if used with its companion receiver the Linx Technologies RXM-433-LR) has a range of up to 750’. Any high output from the Holtek encoder will enable the RF carrier. All keyfobs come set with all address bits tied to ground (bit = 1). The user must float selected address bits (cut their ground connection) to change the address the keyfob will transmit. This is easily done on
the internal PCB, but not necessary unless there are security issues. (You want to select one of the other available addresses.) The fob is offered with up to five push button switches. Any unused data inputs are floating. Buttons are floating when not pushed and pulled to VCC while pushed.

To use this keyfob transmitter with your circuitry, you must also use an RF receiver. The suggested RXM-433-LR receiver costs less than $15 and requires no external tuning components. There are other, less expensive, alternatives like Microchip Technology's rF4020420, but they require a couple of dozen external components. I used the Linx LR in this project, but I might take a closer look at the alternatives for a future column.

**DECODER IN CODE**

Most of your projects will contain a microcontroller. Today, these are inexpensive and can replace other logic and timing circuitry. In many RF projects, communication consists of transferring a serial bitstream to and from UARTs over a RF link. If you were to start from scratch, you could use the common serial data approach. But security issues preclude using this approach because it's easy to duplicate. Using a nonstandard data format makes it difficult for sniffers to determine what's going on.

As you saw earlier, a UART can't decode the Holtek format. It has sync, address, and data information squashed into a single 20-bit transmission. When using a hardware encoder and decoder the format is fixed: 2-bit sync, 10-bit address, and 8-bit data. At the decoder, there really isn't any difference between the address and the data inputs. With a hardware decoder, this 20-bit transmission is treated as a 10-bit address and 8-bit data. Using a microcontroller provides great flexibility. You can choose to emulate the hardware decoder exactly (as in this project) or change the way data is interpreted. Any of the keyfob's incarnations offer less than eight buttons. What happens to the remaining data inputs that are not used? They are not used, and thus wasted. Using a software decoder, you can treat these unused data bits as address bits. This increases the potential receiver pool. My five-button keyfob can add three additional address bits to the available 10 bits, going from 1,024 addresses to 8,196. A hardware decoder would accept multiple addresses because it uses only 8 bits for address matching. You can play around with other possibilities like swapping the address and data bits, which would really frustrate the hardware decoder. But this project's purpose is to show how to implement the hardware decoder in software. So, I'll leave permutation to your imagination.

This project is based on the assumption that you'll do other things with the microcontroller, so the decoder support routines can't tie up the execution. To do this, you need to use two interrupts to eliminate the wait (i.e., inter-bit sampling delays.) Refer back to Figure 3. You know (and can measure) that the bit times for the transmitted format is nominally 2.6 ms per bit, which is calculated in the following fashion: 

\[
(1/76 \text{kHz}[\text{clock speed}]) \times 33[\text{clock divisor}] \times 6[\text{cycles/bit}] \text{ at approximately 3 V.}
\]

Each bit is made from two patterns, each of which is three cycles in length. Each pattern starts with a falling edge (and a low state). One cycle later it can change state. And one cycle after that it must end high. Two patterns make up a bit and can be one of four different types: one, zero, open, or sync.

To decode these patterns you need a peripheral to indicate a falling edge. It can be an external interrupt, a change of state interrupt, a counter/timer trigger, or something else depending on your microcontroller. I chose the Capture/Compare/PWM module (CCP1). The decoding process happens...
automatically in the background and begins with the CCP1 module enabled for falling-edge interrupts. No execution of any decoder code takes place until a falling edge is detected on the DATAIN input. At that point, the CCP1 interrupt is executed and is fulfilling its purpose to clear the SampleState and enable TMR2 interrupts.

During the initialization of the micro, TMR2 was set up with a goal value of 75, which allows it to trigger an interrupt when it reaches counting up to this goal. The goal is one-twelfth of 2.6 ms, or 216 μs. Note that 150 μs after the falling edge is the middle of the first cycle (when the data should be low). When TMR2 triggers, its interrupt routine must first reset TMR2 to start counting again and then look at SampleState to determine which piece of code to execute within this interrupt.

For SampleState=0, you need to sample the data (DATAIN input) to make sure it is low, if not we jump to TMR2_error. Any error will disable TMR2, clear PatternState and MessageState. If there is no error SampleState is incremented and we leave the interrupt.

As long as no errors occur, TMR2 will continue interrupting and SampleState will be increased each time. When SampleState=1, you just increment SampleState and leave.

When SampleState=2, you are halfway through the second cycle where the actual data resides. A second variable PatternState determines whether this is the first pattern in the bit or the second. If PatternState=0, you sample DATAIN and save the state [high or low] into Flags.Data0Bit, increment SampleState, and leave the interrupt. If PatternState=1, then you sample DATAIN and save the state [high or low] into Flags.Data1Bit, increment SampleState, and leave the interrupt. When SampleState=3, you just increment SampleState and leave.

Finally, when SampleState=4, you need to do one more check. You are now halfway through the last cycle of the pattern and the data must be high. If it’s low, it’s an error. If it’s high and the is PatternState=0, you increment PatternState and leave. If PatternState=1, you have completed two passes collecting data and can now determine the found pattern. Using Flags.Data0Bit and Flags.Data1Bit, you set Zbit [zero], Obit [one], TBit [open or tristate], or Sbit [sync] in Flags. After clearing PatternState, you make a call to the AssembleMessage routine and then disable TMR2 interrupts and leave the interrupt. Photo 1 shows timing analysis performed using this code. In particular, notice the encoded transmission and sample points in the top traces.

**DECODE THE MESSAGE**

The hard work is done. You have a way to recognize data bits coming into the micro from the RF receiver, which were transmitted by the keyfob. Each time a whole bit has been processed, the AssembleMessage routine is called to build up a 20-bit message just as it was sent. A third state variable is used for this routine MessageState. Using states are handy especially when the number of states
OVER TWO DECADES OF ENGINEERING EXCELLENCE
At Your Fingertips

CC Gold

The ultimate archive for all things Circuit Cellar: project articles, tutorials, source code, project files, and more.

Loaded with everything you Need to Know about:

- Embedded Applications
- Embedded Development
- Signal Processing
- Wireless Communications
- Programmable Logic
- Embedded Programming
- Measurement & Sensors
- Analog Techniques
- Data Acquisition
- Internet & Connectivity
- Communications
- Robotics
- Graphics & Video

Now you can have the complete Circuit Cellar issue archive and article code stored on a stylish, durable and portable USB flash drive. You can easily keep your CC Gold archive up to date by purchasing subsequent issues from our webshop or by downloading issues with a Circuit Cellar Digital 2.0 Subscription. Issues appear in searchable PDF format.

Complete archive includes PDFs of all issues in print through date of purchase. (For display purposes) USB drive shown is larger than the actual size.

Visit www.cc-webshop.com to purchase
is large. If we need to test for 20 states this could take a long time (20 compares and 20 jumps). Time is critical here because you have less than 150 μs to get through this routine before the next falling edge might occur! So, you must play a dangerous game. You add the value of MessageState to the PC (program counter), which is presently pointing to a list of 20 goto instructions. The program counter is used by the micro to keep track of the next instructor to execute. If MessageState=0, the first [PC+0] goto in the list is executed. If MessageState=9, the tenth [PC+9] goto in the list is executed. This is dangerous. If MessageState should hold an unexpected value, the PC will be positioned to execute from an unexpected location and probably will cause catastrophic behavior. Always test this value and assure that it is compatible before applying it.

You know that every message must begin with two sync bits. So, unless the received pattern of MessageState=0 and MessageState=1 are both sync bits, we have an error. If any of the remaining message bits are Syncs (sync), you have an error. The hardware decoder treats “open” bits as “zero” bits, so we will do the same here (although since we’re the boss with our software decoder, we could choose to make use of this “tristate” as a unique state if we wanted).

With NewAddressL, NewAddressH, and NewData initially cleared, we need only set the appropriate address or data bit when the pattern bit is an Obit (one). If the bit is a Zbit or a Tbit, since its value is “0,” no change is necessary to the corresponding address or data values. After processing the appropriate data bits for MessageState=0-18, you just leave this routine. Each of these visits only cost a handful of execution steps. At MessageState=19, there is cause to celebrate. Assuming the last data bit was received without error, we can finally determine whether this message is for us. If the 10-bit address matches with what you’ve set on the external switches [or hardcoded into the application to eliminate the need for requiring the use of valuable inputs], you proceed to do something with the data. The hardware decoder uses a VALID output bit to signal data is ready. I implemented this and use it as a handshaking device between this routine and the MAIN code execution of the program. If I have a “good message” and data is ready, I check the VALID output. If it’s high [not reset by the user], I just continue on without storing the NewData. If the VALID output is low, I save NewData into OldData and set the VALID output high. OldData is only updated when the user has cleared the VALID output in the MAIN code loop.

**MAIN**

Assuming you have other code that executes in the MAIN loop, a simple test on the VALID output lets you know when new data has arrived from the keyfob. After you do something with OldData, just clear the VALID output and the background decode task will again update OldData and set the VALID output high whenever a “good message” arrives.

What you do with this data is up to you. My first test routine takes this data and places it on the D7:0 output bits. The VALID output pulsates high as a strobe to indicate “good message.”

My application is a bit more complex (see Figure 4) and uses only three output bits to control three external relays based on button pushes. The first relay enables power (D0). The second relay chooses the destination for the power (D1). The third relay reverses the power to the destination (D3). I need the Up and Down buttons to choose one destination [vertical motor]. I use the Left and Right buttons to choose the other destination [horizontal motor]. The Up and Left buttons choose standard polarity [forward]. The Down and Right buttons choose the reverse polarity [reverse]. Up or Down or Left or Right buttons also enable power, while the Center or any combinations of buttons disables power.

To prevent banging a motor with quick reversals, I use a delay between any changes. When new data arrives and any motor is running, the power is first disabled and a delay is executed before continuing. If or when no motor is running, the state of the selection and polarity relays can be changed and then a delay is executed prior to enabling the power relay. Since On is not momentary, the Center (Off) button or any combination of buttons serves the function of shutting all power off to any energized motor.

**IT’S UP TO YOU**

Now it’s up to you to take this and apply it to your own project. I’ve found this keyfob very handy and inexpensive, so I know I’ll be using it in other applications where I need simple wireless remote control of something. The key here is not to reinvent the wheel—or, in this case, the keyfob. And, to make full use of your micro, you should eliminate extra components.

Both Erector and Lionel are still available. Today, railroading uses digital signals to allow all components of complete layout, including multiple trains, switches, and accessories.
to be operated via walk-around controls. Engines/cars now have individually operating authentic movement, lights, sound, and even uncoupling. If you’re unfamiliar with Erector sets, take a look at Figure 5, which is a rather crude depiction of the original “ticker” project I built as a youth. I doubt you’ll get instructions for this in the latest Erector Set.

Jeff Bachiochi (pronounced BAH-kih-oh-kih) has been writing for Circuit Cellar since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or at www.imaginethatnow.com.

**PROJECT FILES**

**SOURCES**
CMD-KEY4-433 Keyfob transmitter, RXM-433-LR receiver, and TXM-433-LR transmitter
Linx Technologies, Inc. | www.linxtechnologies.com

PIC16F737 Microcontroller
Microchip Technology, Inc. | www.microchip.com

---

**Need-to-Know Info**

**Knowledge is power.** In the computer applications industry, informed engineers and programmers don’t just survive, they thrive and excel.

For more need-to-know information about topics covered in Jeff Bachiochi’s Issue 242 article, the Circuit Cellar editorial staff recommends the following content:

**RFID Payment Terminal**
by Carlos Cossio
**Circuit Cellar** 211, 2008
Carlos’s hand-held terminal makes contactless payments a reality. After transactions are complete, the log file stored in flash memory is sent to a host system. Keywords: RFID, Modulation, Antenna

Go to: www.circuitcellar.com/magazine/211.html

---

**Stealth Keyless Entry System**
by David Brown
**Circuit Cellar** 176, 2005
A smart intruder can crack the code to an average 10-button keyless entry pad. David explains how to use your design skills to build a better system. Topics: Keyless Entry, Switch Debounce, Timing, Codes

Go to: www.circuitcellar.com/magazine/176toc.htm

---

**Embedded Systems Development**

*Example:*
State based toy with sound. As each state is exercised, the controller sends an SPI message to the sound generator. Here a sinusoid is used for testing. Both the SPI message, and UART test string are decoded and displayed. We triggered on State 001.

In the USA call:
1-888-75SAELIG | info@saelig.com

---

**Cleverscope**

100 MHz MSO 8M Samples 14 bit

+ Two mixed signal triggers
+ Protocol decoding
+ Spectrum analysis
+ Symbolic maths
+ Custom units
+ Copy & paste
+ Signal generator
+ USB or Ethernet
+ 4 or 8M samples storage
+ 100 MHz sampling
+ Dual 10, 12 or 14 bit ADC
+ Ext Trigger, 8 Digital Inputs
+ 1 MSa/sec charting

www.cleverscope.com

---

*Embedded Systems*
Most embedded systems interact with the real world, and include analog and digital interfaces. State systems are often used for control, and to determine the system response to varying stimuli. Using Cleverscope’s dual triggers you can capture a particular state, and verify that the digital and analog outputs are as expected.
Once More, With Feeling
An MCU + FPGA Without the Compromises

The idea of combining an MCU and FPGA on the same chip isn’t new. Although it looks good on paper, there have been as many misses as hits. What’s the difference that separates winners from losers? Now, Actel is taking a shot at it. Do they have the answer?

Looking back, I can see it was around 10 years ago when I covered the Triscend TES05 (“SoC it to Me,” Circuit Cellar 116, 2000). It combined a fast 32-bit MCU with Xilinx-style (i.e., SRAM look-up table) programmable logic. Around the same time, I wrote an online column covering Atmel’s FSLIC (“Atmel Gets Huge,” CC Online, January 2000), which married their AVR MCU and AT94-family FPGA. Perhaps you even recall when Motorola [now Freescale] made a splash with their “Core Plus” combining a Coldfire 32-bit core with fine-grained SRAM FPGA know-how acquired from UK-based Pilkinson Microelectronics. Or how about QuickLogic and their QuickMPS “ESP” (Embedded Standard Product)? Then, in “Soc Hop” (Circuit Cellar 128, 2001), I covered yet another variation on the theme from a then seemingly unlikely source, the Cypress Semiconductor PSoC. Clearly, the turn of the century was the golden age for integrated processors and FPGAs. So where do we stand today?

Triscend is history. It’s interesting that their demise was practically more newsworthy than their short life. Back in 2004, they were doing the start-up death rattle and there were rumors ARM was going to buy them. Instead, Xilinx swooped in and took control and … nada, the corporate equivalent of taking their ball (FPGA patents) and going home.

FSLIC apparently still lives, at least on the Atmel website. But one has to wonder if it isn’t a “zombie” product, noting for example that the “latest FAQ” [or maybe that’s “inFAQ”] was in 2005.

The Motorola “Core Plus” made a splash alright, except it was a belly flop. It came and went so fast you might have wondered if you’d just imagined it. And if QuickLogic really had ESP [as in extra-sensory perception], they might have foreseen announcing “the end-of-life of our QuickMPS products due to low customer demand.”

On the other hand, the Cypress PSoC is a huge hit. Just ask Cypress founder T. J. Rodgers, not one to mince words, when he says the PSoC “has obviously been a company-changing success” and “is now the flagship product at Cypress.”

There are, no doubt, some lessons here. Let’s see if we, and Actel, have learned them.

ARM & A LEG

Indeed, Actel’s new SmartFusion isn’t even the first chip to combine an ARM core with an FPGA. Triscend had announced one in their heyday, although I can’t recall if it even got beyond the lab and a press release before the curtain came down.

FPGA heavyweight Altera anted up their own ARM+FPGA hybrid with Excalibur [T. Cantrell,