



## HANDS-ON PROJECT: designing microcontrollers with low-cost reconfigurability

RECONFIGURABLE MICROCONTROLLERS OFFER AN IMPRESSIVE ARRAY OF ANALOG AND DIGITAL FEATURES TO MEET YOUR APPLICATION'S REQUIREMENTS.

**M**icrocontrollers are often complete SOCs (systems on chips). Connect power and ground to a programmed device, and you have a powerful processor with active and ready memory and peripherals. These features drastically reduce design time, because using microcontrollers effectively eliminates the job of integrating components and peripherals.

Many of today's 8-bit microcontrollers offer an impressive combination of integrated features and interfaces, including hardware-based ADCs and DACs, PWM support, and a range of serial and parallel interfaces. For small systems, a microcontroller often provides all of the peripherals and interfaces an application requires. The primary limitation of these devices, however, is the quantity and quality of the interfaces. For example, a microcontroller might support a 12-bit ADC on only two of 13 interface pins. If

you need three ADCs or one with 14 bits, you must take a different approach.

This hands-on project focuses on cost-effective flexibility for designing a general-purpose sensing and control module as the foundation for a range of applications. Requirements for the microprocessor included the ability to operate simple user-interface devices, such as LEDs, buttons, dials, and LCDs; to accept data from sensors having a range of data bits and sampling frequencies; and to control analog and digital peripherals, including

In this layout of the PSoC, the buses on the left serve as inputs to configuration blocks and connect to external pins by means of the traces above the blocks. The buses on the right connect to external pins as outputs using the traces below the configuration blocks. Note that every other block is empty in this view; these empty spaces are counter blocks that appear using a different view and have their own connections.

motors and analog-output devices. The design had to support data logging to a centralized location, including both data-collection and -transport mechanisms. It also had to support mesh and hierarchical networks, allowing the ability to tier multiple modules through hub, daisy-chain, or peer-to-peer architectures.

### GAINING FLEXIBILITY

Unfortunately, 8-bit microcontrollers support a limited array of interfaces, and I wanted to be able to support a wide range of applications requiring a varied number of PWMs, ADCs, DACs, and serial interfaces. In my search for a flexible microcontroller, I found the PSoC (programmable-system-on-chip) family from Cypress Semiconductor ([www.cypress.com](http://www.cypress.com)), which employs reconfigurable-logic gates and a switching fabric to define interface-pin functions. PSoC devices cost a little more than microcontrollers of the same functional magnitude, depending on your application.

## AT A GLANCE

Reconfigurable microcontrollers can provide hardware-based processing without the complexity of synthesis.

Through reconfigurability, microcontrollers can overcome interface-allocation limitations typical of fixed-configuration microcontrollers.

Reconfigurable microcontrollers enable a higher degree of flexibility for applications requiring a wide range of I/O options, such as aggregators and expanders.

Reconfigurable architectures have their limitations, but focusing on their abilities enables engineers to cost-effectively extend functions over time.

I was initially wary of looking at a PSoC. I wanted flexible reconfigurability but not at the expense of having to learn a VHDL and synthesis environment to create simple interfaces to LEDs, sensors, and motors. PSoCs, however, avoid this added design complexity. Developers cannot individually access the logic gates within the PSoC; they must instead access them from a functional level. As long as enough resources are available to implement a block, they can define complex combinations of functions and route them to various interface pins.

Designers implement digital and analog functions using configurable blocks. Some blocks perform analog functions, and some perform digital functions. These functions connect through a fabric that eliminates the need to route power, ground, and signal lines among multiple ICs. By allocating additional resource blocks, you can implement complex signal conditioning in hardware, providing high performance without consuming precious processor cycles. You can combine a variety of blocks to create a one-chip mixed-signal design. In the PSoC-development environment, you can select preset functional blocks, such as an 8-bit PWM or a 16-bit ADC, that you want to implement. You can initialize these locked blocks at power-up through registers in flash memory and dynamically

reconfigure them through software APIs (application-programming interfaces), but the PSoC predefines the blocks' basic functions.

This limited configurability, however, is one of the PSoC's greatest strengths. Because the blocks are predefined, I brought up a working PWM in minutes without delving into logic schematics or deciphering preconfigured code. With a bit more work, I was able to quickly and repeatedly replicate the PWM. Given the simplicity of a defined block, all of the API code is consistent and straightforward to use.

## MANAGING CONFIGURATION

I did encounter some difficulties in using multiple copies of the same functional block. I employed a naming convention of PWMx, hoping to make extensive use of macros and subroutines. I found it challenging to reorder and reroute blocks once I had placed them, however. I had placed PWM3 between PWM1 and PWM2 and wanted to rectify this misordering. I had to delete both PWM3 and PWM2 and then redefine

## THIS PROJECT WOULD HAVE NEEDED A COMPLETE REDESIGN TO MINIMIZE LATENCY AND THE AMOUNT OF TIME THE PWM HELD UP THE MICROCONTROLLER PROCESSOR.

them in the proper position to accomplish this task, which I couldn't do easily using drag-and-drop functions. Perhaps, if I had more experience with the tools, I might have been able to do so. However, simultaneously designing hardware and software is a new approach for me, and I had not yet discovered all the tricks and strategies that separate the experts from the novices.

From this lesson, I learned that you must carefully allocate analog and digital resources and interface pins. Early, haphazard placement of blocks can isolate resources in such a way that you may be

unable to assemble them to form a more complex function, or you may lock out the use of certain output pins. For example, I had enough resources to implement an I<sup>2</sup>C interface but not where I needed to—that is, using specific I/O pins because of how I had previously allocated other blocks and functions. Reallocating resources, a rather tedious process, solved this problem, which I could have avoided through planning placement rather than immediately allocating resources.

Another problem I encountered was that the compiler does not support macros within macros, which makes it difficult to create multiple copies of individual blocks of code for each instantiation of a function. For example, a generalized PWM macro cannot call a second specialized macro, significantly reducing the utility of macros. Alternatively, PSoCs support an indirect referencing mechanism that you can use to emulate a record structure for easily passing groups of associated variables and offsets into function vectors. This approach saves program memory but can increase latency because it makes the function indirect and abstract.

The indirect-referencing mechanism, however, took me some work to figure out. The development kit includes a sparse handful of application examples. The memory-allocation model for the compiler has several variations, and none of the applications provide an example of how to successfully declare a variable. Additionally, once I did figure out how to declare variables, I had trouble directly and indirectly referencing them. I partially blame myself for these early difficulties, but the primary cause is the documentation that comes with the PSoC-development environment. The electronic manuals provide little help and no examples of how to declare variables within the memory models. The pseudo-code doesn't work unless you correctly place it within the framework, and the manual contains no reference on how to do so. Fortunately, I discovered a resource outlining ways to avoid many of these headaches that saved me from making time-consuming mistakes (**Reference 1**).

Admittedly, many of these difficulties are mere annoyances rather than fatal flaws. Most vendor-based tools lack adequate documentation. Cypress, for its

part, has a history of addressing the more problematic limitations in updates.

### LIMITED CONFIGURABILITY

Limited configurability is a key element of quickly getting up to speed on a PSoC. The other side of limited configurability, however, is that, once you know what you're doing, you can't modify the hardware blocks. For example, I created several regularly used functions, including ramp-up and -down using the PWM to provide a smooth drop-off. This straightforward function effectively reduces the duty cycle of the PWM. I used a hardware timer and interrupt to implement this function, which consumed many processor cycles and introduced a burst of interrupts.

However, when several PWMs are simultaneously ramping, the load on the processor greatly affects the rest of the application. Additionally, having so many simultaneous interrupts to manage can break an application's real-time determinism. My application had no real-time latency constraints, so I was able to ignore the interrupt contention but still had to struggle with loading on the processor. Alternatively, I considered tying the ramping function of multiple PWMs to a single interrupt to reduce the number of interrupts, but this approach would have impaired my ability to isolate each PWM from an object perspective.

Ideally, I would have liked the ability to implement the ramping function within the configurable fabric, but PSoC doesn't allow you to modify configurable blocks. The function is simple enough and provides the benefit of offloading the processor. Altering hardware-function blocks would have enabled me to further protect my design; burying some intellectual property in the hardware partition makes it more difficult to reverse-engineer a design. This feature is especially relevant if a design must support field upgrades; even if a processor offers hardware-code protection, the code becomes exposed even when it is available only as a binary file. The reality is that the hardware blocks are equivalent to library files for which you lack the source code, which enables you to define and successfully use programmable gates without knowing what is happening inside the chip.

### SOFTWARE CONTENTION

One advantage of working with a programmable mixed-signal part is that a

straightforward partitioning of real-time and application code exists between hardware and software. Even with a fixed-implementation microcontroller, for example, it is relatively simple to implement a PWM. One PWM requires only one timer and integrates well with other system events. Scaling a design, however, is a different matter from implementing a single instance. Using a component method, you can hammer out the details, in this case, of a single PWM. However, the requirement for several simultaneously operating PWMs can be beyond the ability of a traditional microcontroller.

A microcontroller might offer several PWMs, for example, but any application that requires more than the offered number meets a barrier. The next PWM brings with it the cost of implementing it in software. This extra work introduces resistance to entering new markets that require only a nominal incremental increase in capabilities. Adding a second extra PWM is not simply a matter of duplicating the first software PWM. When you implement a PWM in hardware, it has no effect—other than during dynamic reconfiguration—on the main processor. Implementing several PWMs in software raises significant interrupt-scheduling, contention, and real-time-processing issues, because the approach stretches, for example, timer resources to their limits. Furthermore, interrupt handling consumes overhead and introduces synchronization issues—that is, several interrupts can occur simultaneously and require immediate servicing. Also, application code now must compete with high-priority real-time-event processing.

At this point, this project would have needed a complete redesign to minimize latency and the amount of time the PWM held up the microcontroller processor. This design effort is not trivial; furthermore, it focuses on enabling rather than creating an implementation. In other words, I would not be creating any new value, per se, but rather trying to cram working functions into the available space. If the project had used a software PWM from a library, then I might have also had to break into this code to resolve contention issues.

Contention is a serious design consideration that increases disproportionately when you scale a design. As resource sharing within a design increases, so does the

percentage of design effort that you need to focus on simply resolving contention. Even using the PSoC involves issues of managing all of the dynamic changes that require reconfiguration of the PWMs and management of the other functional blocks. An even larger drain on processor and application resources would occur if the design also had to share CPU cycles with the basic PWM function itself.

### MODULE CONSIDERATIONS

This project could have used one of several architectural approaches for the sensing and control module. For example, a module can be either external or a direct implementation on the pc board. I could choose to route I/O interfaces using either switches or generic connectors. Given the integration level of PSoCs, I might even have considered using the chip itself as the entire module; at one point, I reduced a version of my design to a single PSoC and voltage regulator on a breadboard.

Alternatively, I could have created an external module to expand the available I/O interfaces. For example, I worked with the idea of connecting my module to the main system through an I<sup>2</sup>C bus. The module operates as a stand-alone subsystem responsible for all of the I/O that connects to it. The main system has access and control over this I/O through a proprietary protocol that passes over the I<sup>2</sup>C bus.

In the end, the question boils down to just how much flexibility you can take advantage of. Because of the relatively low initial production run of my design, reducing the number of overall designs offers substantial cost savings. For example, I could build a sensing system using a range of components with modules for a number of high-volume configurations and one module to handle all the other configurations. However, each of these optimized modules would require its own NRE (nonrecurring-engineering) charges. At some point, the time and engineering costs to implement several such components becomes excessive, and the implementation would also exceed the available time. In other words, can you spare the 50 hours it takes to shave 50 cents off of a configuration?

This trade-off is a complicated one to consider and depends upon how much time and money you have allocated to increasing product diversity and market share. The advantage of creating a flexible module is that it initially reduces NRE

investment and inventory diversity. Additionally, it reduces the number of interactions with the factory and aggregates overall product volume, leading to a lower overall cost for the initial module. Also, at this stage in a design, you cannot predict which configurations will be high-volume sellers. You can later optimize a module for a configuration if the volumes for it prove adequate. You also don't have to deal with unsold inventory as the market matures; you can repurpose flexible components to sell at the full cost of the applications they can address rather than sell them at a deep discount.

Evaluating the trade-off between potential cost savings of optimizing an architecture for high volumes and maintaining a single base architecture across multiple applications is important. Modules, by their nature, segment functions between software and hardware components. This segmentation can complicate later optimization because modules act independently of other system components. For example, the module for this project had to be a complete and independent system. Therefore, the module must provide enough resources to create a smart sensing system that can capture, evaluate, and transmit data. However, if the sensors for an application require little processing, these resources will sit idle because they are so difficult to make accessible to other system components.

For my project, the fact that I chose to implement 16 interfaces introduced cost inefficiencies. Consider an application that requires 39 I/O interfaces. A 16-interface module can provide 15 interfaces: A minimum of one interface—two for some buses—links the module to other modules, and modules acting as hubs require  $n - 1$  interfaces. Three modules have 48 interfaces, four of which connect two modules to the third, which acts as a hub. This scenario leaves five unused interfaces. Additionally, an integrated approach would eliminate the need for the four communications interfaces. As a result, the device might have to bear the cost of as many as nine more interfaces than the application requires.

Another key source of both increased cost and increased unreliability is the need for a generic connector. The use of a generic connector extends the flexibility of a module. If I could guarantee that any application using a module would require a minimum number of LEDs,

motors, or I<sup>2</sup>C ports, I could have dedicated the appropriate number of interfaces and connectors. However, in many cases, the ability to cluster components on a single module lets you increase the intelligent processing on the module itself. For example, consider a system that requires 14 sensors and 14 LEDs. For one application, the LEDs display the status of each sensor. An LED that is continuously on means that the sensor is active, a flashing LED means that the sensor is experiencing an event, and an LED that is rapidly flashing means that the sensor needs servicing. For this application, it makes the most sense to group each LED with the appropriate sensor—seven sensors and seven LEDs per module with two I/Os reserved for a communications link.

On the other hand, the LEDs may convey information about the overall system based on aggregated processing of multiple sensors. For example, a group of sensors might monitor the same system component from different locations. Determining the status of the component requires using data from all of the sensors. If the group of sensors resides on two modules, then the modules may need to exchange a significant amount of data—introducing unwanted latency, as well—to resolve the status of the component. For this application, grouping the 14 sensors provides the most efficient means of intelligently managing the sensors. The sensor module would need to transfer only the state of each LED to the LED module. From a logical point of view, the LED module serves as an I/O expander for the sensor module, which itself is an I/O expander for the main system.

If the module configuration has constraints, such as a minimum or maximum available number of ADCs and LED drivers, more modules are necessary to provide the proper number of appropriate interfaces. Most traditional microcontrollers employ this architectural model. Although each microcontroller-product family offers a variety of interface configurations, these configurations have limited scope and scale. Given software-implementation limitations, it might require more microcontrollers than reconfigurable devices to implement a configuration.

With the availability of low-cost reconfigurable or field-programmable devices,

MORE AT EDN.COM



+ We encourage your comments!  
Go to [www.edn.com/060330df1](http://www.edn.com/060330df1) and click on Feedback Loop to post a comment on this article.

designers of embedded-system applications, such as industrial-control, motor, LED, and sensor-management systems, can now implement functions in hardware rather than software, enabling developers to both consolidate system management in fewer processors and increase device flexibility through easily and dynamically adjusting hardware interfaces.

The primary trade-off occurs when you decide how much flexibility and what range of functions you need. This decision determines how closely a fixed-microcontroller architecture matches your application. For applications requiring two ADCs and a lot of general-purpose I/O, for example, an off-the-shelf microcontroller offers you a more integrated and less complex foundation. When the I/O of a design may change over time or an application needs more instances of a set function, such as a PWM, than microcontrollers typically offer, then a reconfigurable architecture offers a clear advantage.

Some engineers are pessimists. They look at a project and note all the ways they cannot use it. I tend to be optimistic. I understand that every product has its limitations, but what matters most about each product is what new innovations it enables me to create. A PSoC as a reconfigurable architecture offers many features that are beyond the scope of this project. However, it is exciting that PSoCs offer the flexibility of substantially extending my design simply by cracking into a new set of modules. EDN

## REFERENCE

■ Ashby, Robert, *Designer's Guide to the Cypress PSoC*, Newnes Publishing, 2005.

## AUTHOR BIOGRAPHY

Nicholas Cravotta is a contributing technical editor for EDN. When he isn't designing electronics or writing, he enjoys building kaleidoscopes and developing board and card games. You can reach him at [editor@nicholascravotta.com](mailto:editor@nicholascravotta.com).