Balancing Performance and Power Efficiency in Embedded Systems

Introduction

Optimizing embedded systems for low power consumption requires developers to find a balance between performance and power usage. However, achieving this balance can mean compromising product capability and reliability. There are three areas in which these compromises can affect performance: analog sensing, communication and algorithmic processing. For example, a serial communication interface may require cumbersome protocol modifications to compensate for “dead time” communications failures when the microcontroller (MCU) is in a deep-sleep mode. An embedded design with analog capacitive sensing inputs might not be as robust against interference or moisture because the sensor’s scan time must be reduced to save power. Some algorithmic features of a design might be made less feature-rich because current consumption limitations require a reduction in active mode processing time.

Embedded developers can mitigate the effects of these compromises or avoid these effects altogether by choosing a 32-bit microcontroller that is well suited for high-performance, low-power applications. Ultimately, specific firmware optimizations that take advantage of the MCU’s capabilities can help maintain the system’s performance and reliability while reducing current consumption.

Embedded Design Example: IR-Based Remote Control

Consider, for example, how an infrared (IR) remote control product will benefit from low-power optimizations. Let’s assume that the remote control contains the following features:

- A serial interface with an onboard IR transceiver
- A user interface consisting of four capacitive sensing buttons
- Some consideration toward “future proofing” the design for reuse in future, more feature-rich products

Create a Power Budget

MCUs operating in low-power applications follow a few basic principles (see Figure 1):

- Keep the MCU in its lowest power, deep-sleep mode whenever possible.
- When performing a task, firmware should tend toward the lowest power operational state possible.
- Process-dependent tasks, such as the execution of a digital filtering algorithm, should use the fastest clock speed feasible to meet all other design requirements.
- Time-dependent tasks, such as receiving a byte of data across a serial interface at a defined baud rate, should use the slowest clock speed feasible in the design.

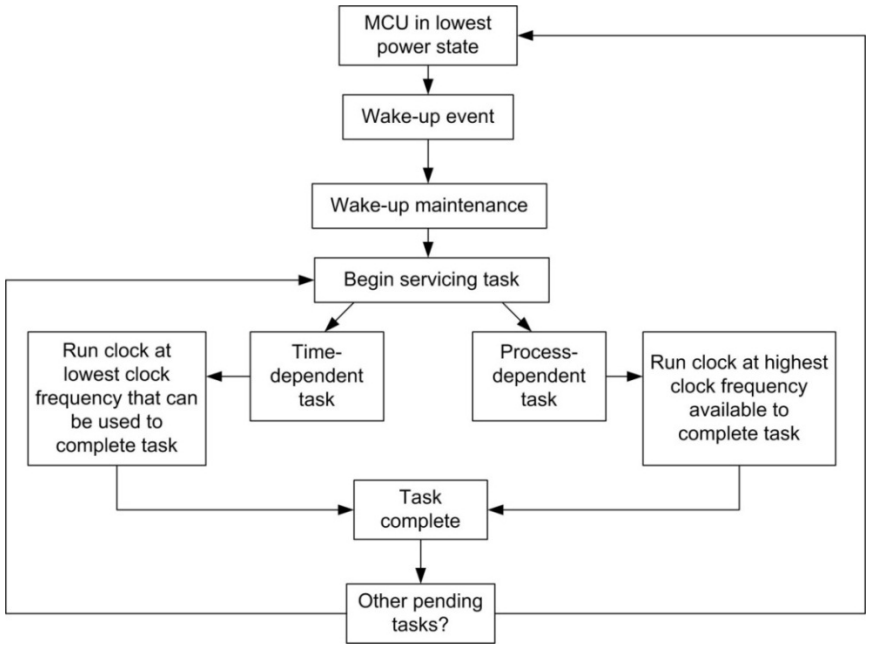

Figure 1. Example of Low Power Execution with MCU

The developer should break down the system's firmware architecture into functional components and try to determine which components are process-dependent and which components are time-dependent. In addition, it is helpful to try to estimate the time required to execute each firmware component and the frequency at which each component will execute. Based on these estimated values, the designer can generate a rough estimate of the average current consumption.

Creating this preliminary power budget helps designers in a few key respects:

- Designers can gain an understanding of the type or types of batteries that will be appropriate for the design, which influences hardware considerations and could save hardware revisions during development.
- Developing a more accurate estimate of current consumption requirements enables designers to make more intelligent decisions regarding which microcontroller is best suited to the design.

The remote control application example has the following functional blocks, as shown in Figure 2:

- IR serial interface – a time-dependent task because the serial interface will operate at a defined baud rate
- IR command parser – a process-dependent interpreter algorithm
- Capacitive sensor to detect touches on the sensing electrodes – a time-dependent task because the capacitive sensing block will require a defined time or range of time to take a sample
- Robustness-enhancing algorithms to filter the capacitive sensing output and prevent false touch detection – process-dependent task requiring MCU cycles and resources to complete

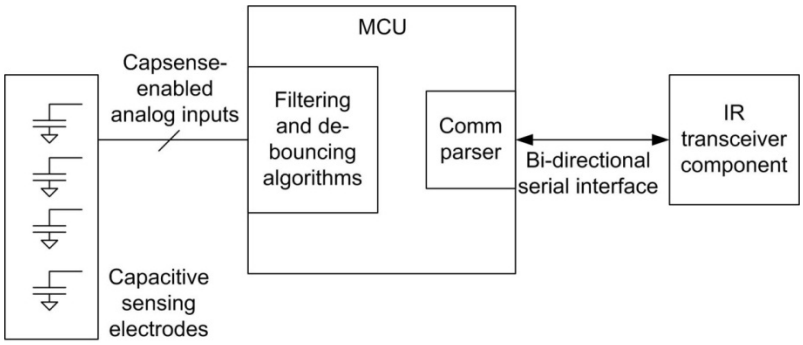

Figure 2. Basic Functional Blocks in an IR-Based Remote Control

Choosing the Optimal 32-bit Microcontroller

When choosing a 32-bit microcontroller for an embedded system, designers must dig deeply into product specifications. Most MCU vendors offer product selector guides and matrices that show basic feature sets, code and RAM footprints and pinouts of the MCUs they offer, which are useful first-pass tools to eliminate MCUs that do not meet basic design requirements. However, be cautious when comparing feature sets between MCU vendors, as specifications from one vender might be defined differently by other vendors.

For example, if a design requires a high-performance analog-to-digital converter (ADC), make sure that the ADC achieves the level of performance you require across the temperature and voltage range that matches the operating range of your product. Some integrated ADCs cannot achieve their highest performance specifications across the full operating range of the MCU. Careful specification examination is especially important in low-power applications. Some MCUs will be affected by multiple performance compromises when operating at the lower voltage levels seen in battery-powered applications.

In addition to checking high-performance analog peripherals for specification compromises, make sure to check that the microcontroller can operate at the system clock speed your product requires. An MCU that can only operate at 75 percent of the expected oscillator speed when operating on battery power will have a dramatic impact on average current consumption in systems with process-dependent firmware components.

One microcontroller capable of delivering the performance required without exceeding our low-power current consumption requirements is Silicon Labs' SiM3C1xx Precision32™ 32-bit microcontroller based on the ARM® Cortex™-M3 processor. This 32-bit MCU includes a sophisticated power management block designed to achieve low power performance, an on-chip 16-bit capacitance-to-digital converter, many serial interfaces and a code/RAM/feature set that gives enough performance margin to be future-proofed and ready for reuse in more advanced applications.

Using a Real-Time Clock with Capacitive Sensing

The SiM3C1xx 32-bit microcontroller devices' capacitive sensing block can perform a 16-bit conversion in less than 50 microseconds. The sensing firmware component does not need to constantly perform conversions to determine whether the user has pressed a button. Instead, the firmware can take advantage of the MCU's on-chip real-time clock feature to periodically wake from a deep sleep and perform a scan.

Because crystal-enabled accuracy isn't a requirement in our example application, we can operate the clock in a self-oscillating mode that does not require a discrete crystal oscillator. By running without a

crystal, the system can conserve the current that would be required to excite and oscillate an external crystal.

Optimizing Initialization to Minimize Reset Time

The SiM3C1xx 32-bit microcontroller's ARM processor core uses Power Mode 9 (PM9) to achieve its lowest power operational state. The real-time clock can be configured to force a wake-up when the clock's alarm trips. The MCU must reset at the beginning of this wake-up process, which can lead to unnecessary execution time if firmware is made to re-initialize all hardware peripherals and variables. To save processing time and conserve current, firmware should check the "wake-up source" immediately after reset to determine why the device reset.

Based on the reset source, firmware can "fork" initialization to save time. If the reset was caused by a power-on event, as would be the case if the product's battery were replaced, then complete initialization might be needed to establish a defined start-up system state. However, if a real-time clock alarm caused the reset, then firmware may only need to initialize the capacitive sensing block to execute the scheduled conversion. If processing determines that the user is not touching one of the electrodes, it can then quickly return to a low-power PM9 state and avoid extraneous code execution. Figure 3 shows the cycling flow for a typical PM9 reset.

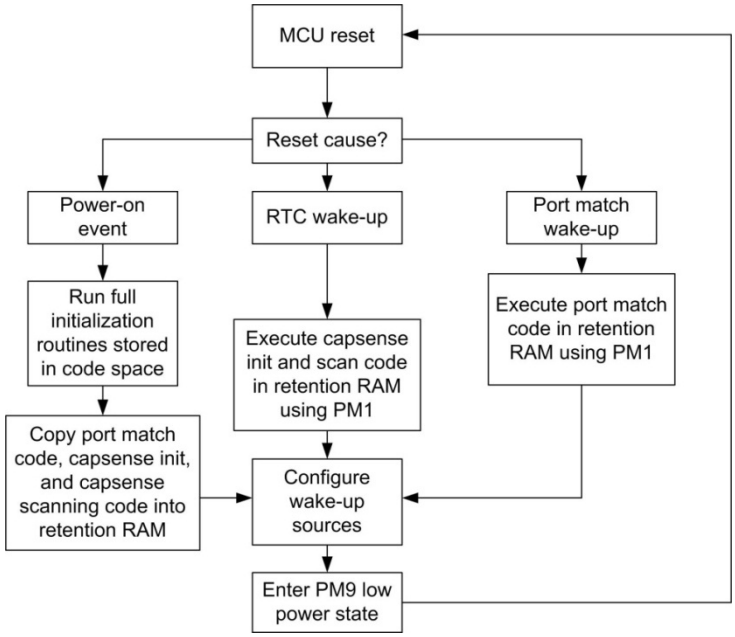

Figure 3. PM9 Reset Cycling Flow Chart

Power Considerations During a Conversion

The capacitive conversion is time-dependent, and thus firmware should try to revert to a lower power state during the conversion to conserve current. In the case of the SiM3C1xx 32-bit microcontroller, a conversion can execute while the MCU is in a Power Mode 3 (PM3) state. The MCU will automatically exit this state and resume operation when the conversion completes.

Any robustness-enhancing algorithms that need to be executed on the newly sampled data should be run on the fastest available system clock to complete this process-dependent task as quickly as possible.

Using Port Match Features for Wake-on-IR Reception

The SiM3C1xx 32-bit microcontroller can also wake from PM9 when it detects a transition on a digital port pin. This feature enables the system to wake when the IR transceiver begins transmitting data bytes across a serial interface. The port match feature can watch an interrupt pin driven by the IR component, or it can monitor one of the interface's data or clock lines.

Keep in mind that the MCU's serial interface will not be enabled and configured to receive data immediately after a PM9 wake-up because the device must cycle through a reset. This delay can be minimized using the initialization forking optimization feature.

More Serial Interface Power Considerations

The time- and process-dependent portions of the IR transaction should be handled just as similar tasks were handled in the capacitive sensing block. When possible, the system should go into a low-power state. When algorithms need to be executed, make sure that the system is using the fastest system clock possible.

Power Mode 1 Execution

Low-power microcontroller firmware can do even more to conserve current. For example, the SiM3C1xx MCU can execute code from RAM instead of code space while in Power Mode 1 (PM1), which conserves current because no flash memory fetches are required for instruction execution. When firmware pairs this feature with the use of retention RAM, which maintains state across the reset cycle, both capacitive sensing and IR interfacing can achieve optimally low current draw when executing.

Conclusion

In many respects, architecting a low-power system is an exercise in "doing more with less." At every point in the development process, from MCU selection to code creation, designers should be asking themselves whether there is a way to avoid performing unnecessary tasks. When the task must be performed, the driving priority is to complete the task in a way that achieves design requirements while consuming as little current as possible. However, designers must take care that optimizations made to the system do not severely compromise performance.

In the remote control example, for instance, performing capacitive sense scanning too infrequently could result in missed button presses or insufficient robustness. If the IR interface is not serviced at an adequate speed, IR commands might be ignored or misinterpreted. Finding the right balance between performance and power savings is a challenge that can be overcome with careful planning and design principles that ably execute all requirements with judicious current consumption. Find out more about Silicon Labs' 32-bit microcontroller devices at www.silabs.com/32bit-MCU.

# # #