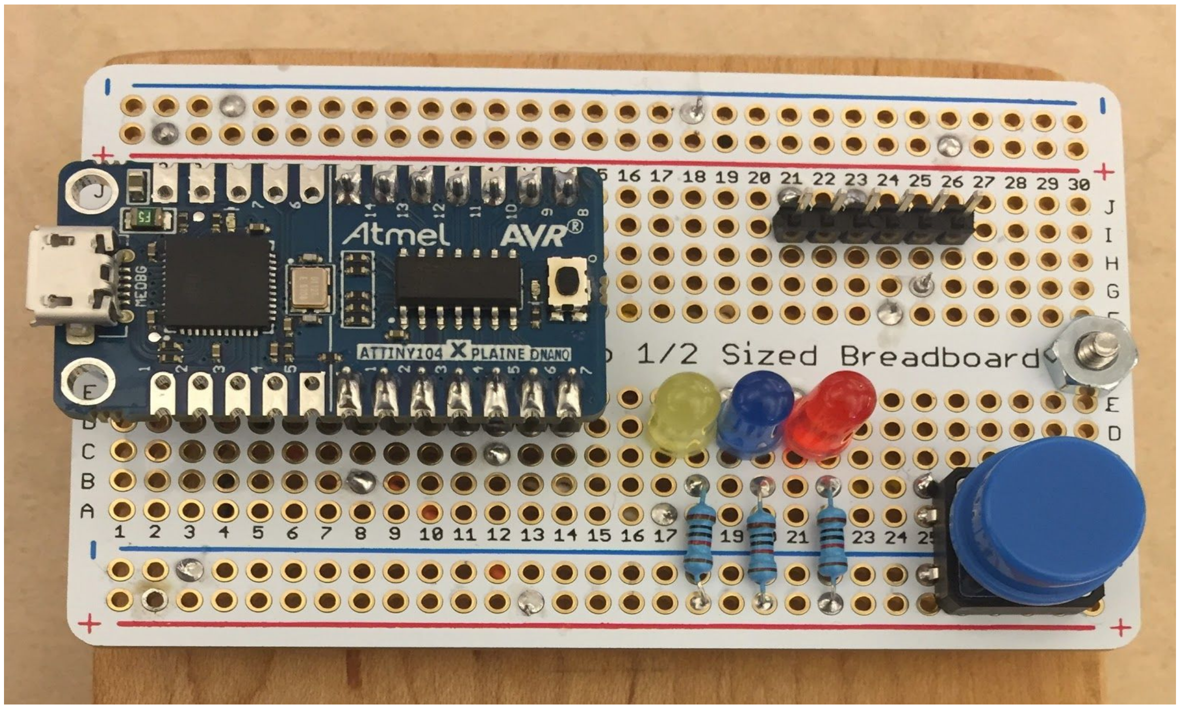
Secaplius

Architectural Pattern for a Very Small Microcontroller

September 12, 2016



Introduction

Using a standard architectural pattern for a very small microcontroller.

When I am asked to create a software application for a small 8-bit microcontroller, I usually default to an architectural pattern that has served me well for the last 8 years. Typically, these smaller microcontrollers have 8K to 32K bytes of Flash memory and 1K to 2K bytes of RAM. Many 8-bit microcontrollers are too small to make use of the standard RTOS implementations, but large enough for this simple task scheduler.

This spring, Atmel introduced the ATtiny104 microcontroller and I asked myself the Limbo question. How low can you go? Is it reasonable to use a task scheduler on a microcontroller with 1K bytes of Flash and 32 bytes of RAM?

ATtiny104

The ATtiny104 hardware implements a variety of modern peripherals and Atmel has great development tools, free, as part of their Arduino effort. Even if the ATtiny104 is impractical for my use, I still have a nice IDE and compiler to experiment with at home.

How to Measure Success

Criteria for determining if a resulting design is useful.

Many small microcontrollers are rejected for use in high reliability systems (medical, automotive, etc.) except in low risk portions of the overall device design. Their program space is too small to include both the application and failure mitigation software components. If the microcontroller is only suitable for low risk applications, it will be rejected in favor of a larger more expensive device that is suitable for design reuse.

My needs require that the base software application include watchdog management and a rudimentary power on self test. Task execution must be deterministic and tasks should have a mechanism for failure reporting. Also, the ATTiny104 is designed for battery powered devices so the application must contain a low power management access point.

Application Description

Components included in the implemented architectural pattern.

I limited the base application size to one third of the available Flash memory. I was able to implement most of the required functionality, but I was not completely satisfied with the result.

Watchdog Management - The watchdog was easy to configure and a watchdog trap was implemented to prevent the application from running if a watchdog reset is detected.

Power On Self Test - I tried to implement both a RAM and Flash memory self test and included a disable flag for POST to enable debugging. I eliminated the RAM self test to keep the size under the 30% limit. The Flash memory checksum code was easy to implement, but it requires that the output hex file be modified after the build to add the checksum into the last two bytes of Flash memory. The example programming file contains fill bytes equal to 0xFF in unused memory (always a good practice).

Timer Driven Task Scheduler - The task scheduler was easy to code, because the timer interrupt service routine was very simple to implement. The final code structure was very similar to the version used for larger applications except that bit flags were used to signal when tasks are ready to run and the scheduler required direct calls to the task modules task functions. A larger application would utilize task function pointers to eliminate coupling of the task scheduler to task modules.

Failure Reporting - The only failure reporting implemented is the on-board LED used to indicate a watchdog reset or the failure of the Flash POST. The space available for failure reporting is insufficient for high reliability designs that require an exception handler to report failures, save failure history and to limit application execution when critical failures occur.

Low Power Management - I was successful in implementing a sleep cycle after all tasks have been serviced for each timer tick of the scheduler. The example application executes very quickly at 8Mhz so the CPU sleeps about 97% of the time. Additional sleep routines would need to be added for battery applications that need to conserve power for long periods of inactivity.

Description of the example application.

Three task modules were created for this application to implement a simple user interface for a button and three LEDs. Long and short button presses are detected to allow the user to change the selected program for device operation (Program 1, 2 or 3).

The application is just an example for demonstration. Single button interfaces with long, short and repeated button presses are not intuitive to the user and in my opinion should be avoided.

LEDtask - This task will initialize the LED outputs and flash an LED for a specific number of flashes in response to a call to the interface task “LEDFlash(uint8_t flash_count, T_LED_NAME color)”.

Buttontask - This task will initialize the button input and other tasks can poll for button events “T_BUTTON_EVENT ButtonEvent(void)”. Normally, I would use a callback for events, but RAM size limits this type of implementation.

Modetask - This task implements the program selection behavior. I will not explain how this state machine operates as it is not important for this discussion.

Conclusions

How would I use the ATTiny104 microcontroller?

The ATTiny104 CPU and peripherals are impressive, work as expected and the device is incredibly flexible. The microcontroller is well suited to a specific single purpose consumer application like an electric toothbrush or kitchen appliance. I would consider the device for test fixtures or non-critical applications. Flash memory size and RAM size are so small it would be difficult to add the safety and reliability features needed for a critical automotive or medical application.

What did I like?

Hobbyists will love this device. I was able to create a development board for less than 20 dollars and implement the application with free tools. The Hex Workshop tool for Intel hex file edit was not free, but I think there are free hex editors available. Atmel Studio 7.0 was easy to use, but I have used many similar IDE tools.

What difficulties did I encounter?

I assumed that the MEDBG chip on the ATTiny104 XplainedNano development board would provide some limited debug features for software development, but this is not the case and developers will need to revert to the techniques of yesterday (output toggling, etc.) to debug applications. The interface chip allows the user to program the ATTiny104. I did try to do some digital signal filtering and serial communications, but 1K byte of flash is very limiting for anything that has the sophistication to be truly usable.

Appendix

Application and Tools

I apologize for leaving out many of the details of the hardware prototype design and the use of Atmel Studio 7.0 for software development. If the reader is interested, most of this information is available from the Atmel website.

IDE and Compiler - Atmel Studio 7.0 from Atmel.

Editor for Flash checksum insertion - Hex Workshop v6.7 from BreakPoint Software.

Embedded development platform - Atmel ATTiny104 XplainedNano (purchased from DigiKey).

Half sized prototype board - Adafruit (very nice proto board for a reasonable price).

Application - The application file “Prototype1.zip” can be unzipped into the directory: “C:\Projects\Atmel\ATtiny104\Proto1\Prototype1”. The Atmel Studio solution file is “Prototype1.atsln”.