Summary

In this article i will explain some enhancements on AVRILOS SysTick module, as long as a few new features that you will find them useful.

Introduction

AVRILOS is a simple task co-operative kernel with a small footprint. See relative links on the bottom of the article to find more about AVRILOS.

Background

One of the basic modules of AVRILOS is the SysTick timer. This timer keeps timer housekeeping for the OS (ie. trigger ADC sampling, generates delays etc) while on the other hand can help the application to do some time functions like delays or timeouts.

However on one of my latest project i was faced with a failure to keep timing correct when i modified the various tick constants on ifc_time.h (c_ALIVEOK_ms in this case). I noticed that the LED Alive LED was flashing with a non monotonous flashing. This was so obvious (but not critical) that i would like to investigate further and see why this happens. Also i found a little difficult to define properly the various ticks (application, ADC) if i needed to modify the relevant constants.

Another issue i had, was that i needed to have another LED flashing as the Alive one from my application. I had a second LED attached to a different port and physically was out of the box, so it was visible by the user. The issue there was that there was no provision for such functionality in the OS so i had to hack the SysTick in order to perform this task. So a more proper way was introduced.

This particular application (pump control) needed long-run timers (hours/days), so i added this functionality in SysTick as well.

Given a small issue on the previous article AVRILOS & FPGA where we were controlling R/C servos from the FPGA, I split the application function in two flavors. One which runs every application tick (c_AppInterval_ms) and one which runs continuously on the main loop without waiting this interval. Thus you may implement whichever mechanism you want (or even both).
Finally i will present the small pump application i did on summer and demonstrates most of the above functionality.

Description

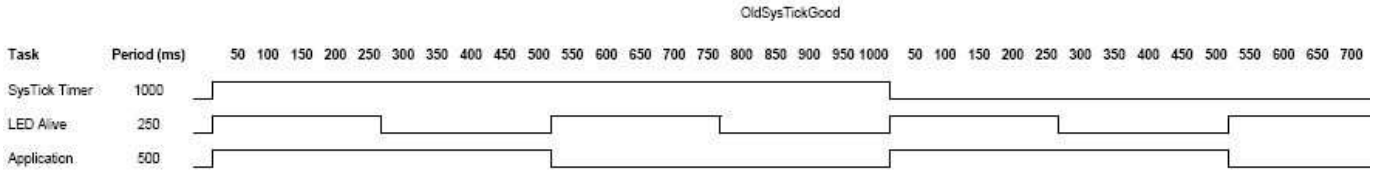
Changes: Enhancements of old functionality - Bug Fix

The old SysTick used up to now used a modulo check to see if each task required to run at relative intervals was ready. Below is a code excerpt from SysTick that shows the module check for LED Alive.

```
if ( (v_SysStat & (1 << b_SysTick ) ) != 0 )
```

```
{
        v_SysStat &= ~(1 << b_SysTick );
        v_SysTimer++;

        // Wrap around Timer
        // Interval is 1 second
        if (v_SysTimer == c_SYSTIMEMAX) v_SysTimer = 0;

        // Toggle AliveLED depending on error state
        if ( (v_SysTimer % v_ErrLevel) == 0)
        {
            f_flashled();
        }
}
```

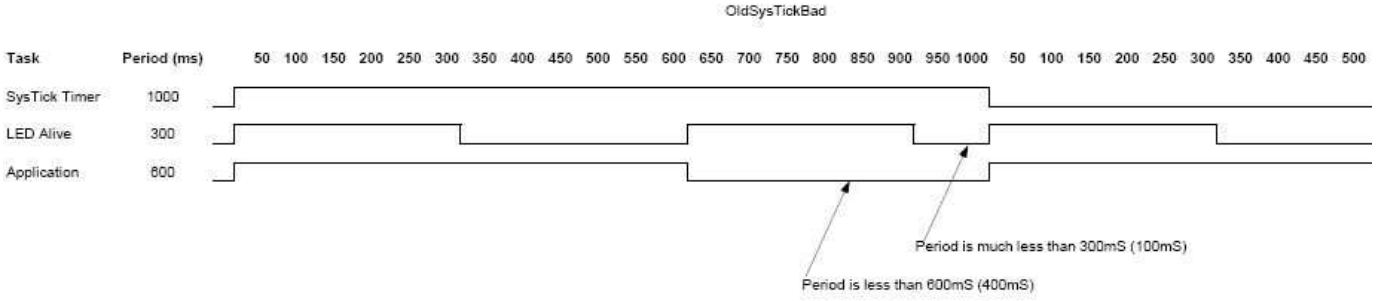
The SysTick Timer runs from 0-999mS. Then checks with the period time using the division remainder. In the following diagram you see the normal scenario.



This method have the advantage of using only one variable (The actual SysTick Timer) in order to execute many different tasks with different periods. Although this seems elegant (or cool) it has some limitations (or bugs...).

If the various tasks (ie. LED Alive, ADC, etc) periods do not have a maximum common divisor that is an integer factor of the SysTick timer duration, then abnormal periods can be generated which are systematic but not monotonous. See below diagram for a "bad" case, where the periods selected do not match the total duration of 1000mS.

The problem is revealed when the master SysTick counter is wrapped around. Then the remainder of SysTick counter (0) is zero in any case. Thus on the event of SysTick wrap around the problem occurs.



In order to resolve this issue i had to add one variable per task (and thus use a little more RAM). Then each task has its own counter for measuring period and thus are completely in-depended and accurate. Also the ifc_time.h include was modified to provide correct timing definitions for this case.

Old code was:

```
#define c_T0Unit_ticks_per_ms    (MPUCLK_Hz/c_T0DIV)
/c_ONESECOND_ms
#define c_T0TimeSlice
256-c_T0Unit_ticks_per_ms           // for 1mS interval
#define c_TimeSlice_Hz           c_T0TimeSlice/(MPUCLK_Hz
/c_T0DIV)

#define c_SYSTIMEMAX             c_SysTickPeriod_ms
#define c_ALIVEOK_ticks          (INT16U)
(c_TimeSlice_Hz/c_ALIVEOK_ms)     /* Alive LED When Ok */
#define c_ALIVESER_ticks         (INT16U)
(c_TimeSlice_Hz/c_ALIVESER_ms)    /* Alive LED When Serial
Error */
```

while the new code is:

```
#define c_T0Unit_ticks_per_ms
(MPUCLK_Hz/(c_T0DIV*c_ONESECOND_ms))
#define c_T0TimeSlice
(256-c_T0Unit_ticks_per_ms)
#define c_TimeSlice_Hz              (MPUCLK_Hz/(c_T0DIV *
c_T0TimeSlice))

#define c_SYSTIMEMAX               c_ONESECOND_ms
#define c_ALIVEOK_ticks            (INT16U)
(c_TimeSlice_Hz*c_ALIVEOK_ms/c_ONESECOND_ms)     /* Alive
LED When Ok */
#define c_ALIVESER_ticks           (INT16U)
(c_TimeSlice_Hz*c_ALIVESER_ms/c_ONESECOND_ms)    /* Alive
LED When Serial Error */
```

Another problem on the old code was that c_TimeSlice_Hz was not properly defined (note the relevant parenthesis on new code) and thus timing definition would be wrong in many cases.
This is interesting because i have never changed these parameters and I have never seen this problem before. Actually i discovered the problem looking at my LED Alive period. I then noticed that i could not change in a deterministic way the flash period (wrong definition) and also i saw a glitch due to common divisor inconsistency.

At the expense of a few more variables the problem is solved. See below code fragment of SysTick.c:

```
        if ( v_SysAlive > v_ErrLevel)
        {
            v_SysAlive = 0;
            f_flashled();
        }

        if ( v_SysApp > c_APP_ticks )
        {
            v_SysApp = 0;
            v_SysStat |= (1 << b_AppTick );
        }
```

Thus each individual task uses its own variable independently.

Changes: New Feature - Long Timers

In my presented example application i needed timers that could count hours or days. My existing SysTick timer module did not have this capability. In order to accommodate this functionality i

have added a number of timers (number of counters defined in ifc_time.h).

The new name of the long timer variables is: v_SwLongTimer_mS[]. It is an array (ifc_time.h defines how many timers you need). It counts down (like the software timers defined in SysTick) until they reach zero (completion of counting). The counter countdown is performed every 1mS (1 SysTick period). Thus we can measure 4 billion mS which is a very long time. These timers are used for Timeouts, which allow for easy implementation for such operation as seen in the first AVRILOS article.

Of course for more accurate timing you should use the AVR's 32KHz crystal used for real time clock. However in my hardware i did not use the extra timer so this is an alternate way to do it.

Given that crystal oscillator frequency may not fit exact division for seconds (depending from the prescaler values) you will have small errors accumulated over time. For the pump application this was not a big problem. Otherwise you can trim down the initial count down value to provide an accurate timing.

The code that handles the long SW timers is provided below:

```
        for(v_idx=0;v_idx<c_MAXSWLTIMERS; v_idx++)

         {
             if (v_SwLongTimer_mS[v_idx] != 0)

             {
                 v_SwLongTimer_mS[v_idx]--;
             }

         }
```

Each non-zero counter is decremented until zero. The application check to see if the counter cleared to determine expiration.

Changes: New Feature - AliveHook

Then I had another issue! I had to reflect LED Alive status to a different LED connected to another I/O. The second LED was visible from outside the box. However I had to hack the Systick to provide this functionality. As i wanted a more permanent solution i decided to make a weak function call to a hook function:

```
void f_flashled(void)

{

    INT8U v_temp;

    v_temp = (c_PORTALIVE);
    v_temp ^= 1 << b_LedAlive;
    (c_PORTALIVE) = v_temp;

    f_HookAlive( ((v_temp) & (1 << b_LedAlive)) );

}
```

I call the f_HookAlive inside flash LED function. The parameter signifies if the Alive LED is on or off, so there is not need for

housekeeping toggle code. This is performed already in f_flashled.Then on my application i do the following:

```
// copy led alive to external LED
void f_HookAlive(INT8U v_ledstate)

{
    if (v_ledstate == 0) LedYelOn;
    else LedYelOff;
}
```

SysTick module already has an extern reference to this function so the linker will have the connection and will call this function.

However what happens if the hook is not defined? Well the small secret is the weak definition in SysTick:

```
extern void f_HookAlive(INT8U v_ledstate)
__attribute__((weak));
```

Defining the function weak means that this function may not be present. The linker will ignore the reference to a non-existent weak function (no-call) and everything goes smoothly. In case you need the function you only have to declare it somewhere in your program. In this case the linker will see the connection and will call your function.

Changes: New Feature - Application Timed & Cont. operation

In the previous article the R/C servo driving was more smooth when the application did not run every application interval c_AppInterval_ms. This had to due with the combination of UART and the Servo control. there were many keys accepted before the Servo control ran. Thus the program were executed user input in batches. The solution was to remove the application from the timed trigger section in the Kernel.c:

```
            if ( ( v_SysStat & (1 << b_AppTick) ) != 0)

            {
                v_SysStat &= ~(1 << b_AppTick);
                //f_Applic();  // Initial placement of
application
            }
            f_Applic(); // R/C Servo placement for smoother
operation
```

I had then the idea that we could provide such versions on the standard empty application and the programmer could use whichever hook he or she liked (or even both).
The kernel.c modified as follows:

```
            if ( ( v_SysStat & (1 << b_AppTick) ) != 0)

            {
                v_SysStat &= ~(1 << b_AppTick);
                f_Applic_Tick();
            }
            f_Applic_Cont();
```

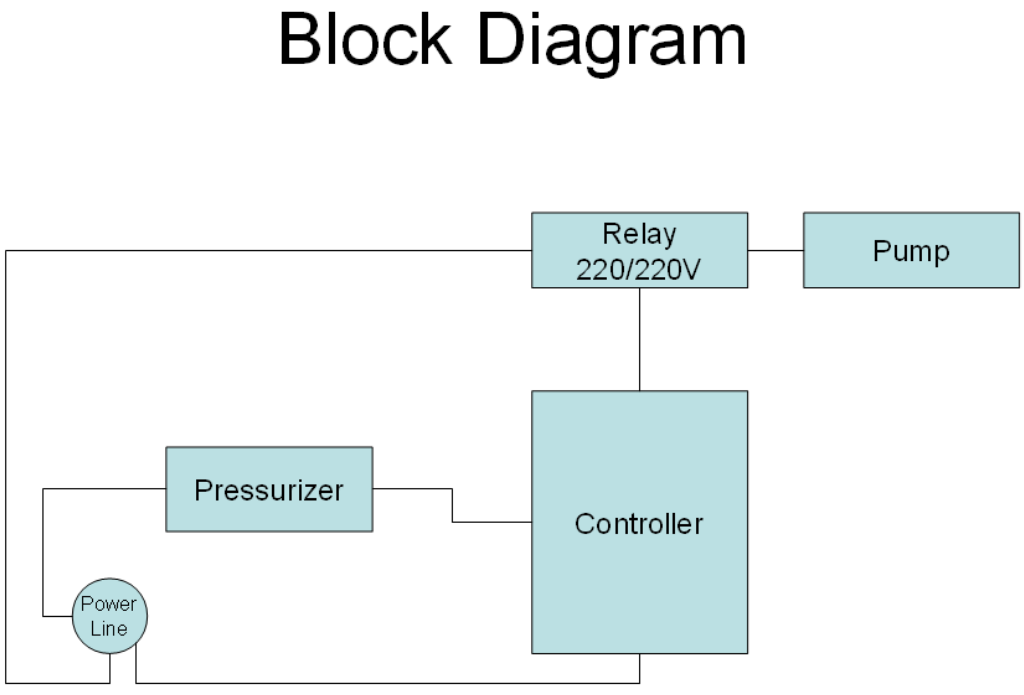
Then Applic.c modified as follows:

```
// Function called by the main kernel loop (task)
// Should not block
// This process runs on every Application Tick
// Use this for timed event handling
void f_Applic_Tick(void)
{

}



// Function called by the main kernel loop (task)
// Should not block
// This process runs continuously on the main loop
// Use this if you need no wait time on each run
void f_Applic_Cont(void)
{

}
```

Now it is up to the programmer to use whichever version he needs depending on the application.
The aim here is to not modify the Kernel.c for application specific reasons but only for new device drivers.

Example: Pump Control

In the example application i have the following pump system:



Physical arrangement is as follows:
The pressurizer wants to keep water pressure above a minimum limit. When this low limit is reached requests the pump to activate. Normally the pump fills up the pressurizer water tank and after a high pressure limit is reached the pressurizer stops the pump. Thus we guarantee a minimum of water pressure on the

output of the pressurizer.

This is a classic automation used for gardens. The problem is that if there is a leak somewhere the pump may work for too long and may be there is a flood or a burned pump.

The target here is to allow pump normal operation but stop the pump in case of abnormal activity. For example there are two scenarios:

1. The pipe from the pump to the pressurizer is broken.
2. Some pipe after the pressurizer is broken.

Case 1:

In this case the pump will fail to fill the pressurizer. The pressurizer requests pump activity to fill. The leak will be large and the whole garden could be full of water. The key here is to stop the pump after a maximum time that the pressurizer would have filled up (with some margin). This is the continuous operation limit; ie the pump must not exceed continuous operation above a limit.
Normally this limit should be in the minutes range (ie. 4 minutes).

Case 2:

In this case the pressurizer fill and empty too often. The total pump time during a day should not exceed a reasonable limit otherwise there would be some leak. This is the daily limit. If any such limit condition occurs then system stops operation until there is a switch press (or in my case a photo-resistor detects case open of the control box - this is the general reset). For example in my case this limit was about 2 hours daily operation.

Opening the case bypass the limit & counter operation. This is achived by freezing the internal counters. The code uses a very simple state machine. It sits in the IDLE state until there is a pump activation.Then it jumps to WORK state where the counters are incremented. Then it can either return to IDLE or if the one limit was violated jumps to BLOCK state (pump off). Then a case open will reset the state machine back to IDLE. Code is small and self-explanatory.

Conclusion

In this article i presented some bug-fixes and enhancements regarding the timing functions of AVRILOS.

In my next article i will present Mercurial (hg) access to the AVRILOS repository in SourceForge.

Relative Links

avrilos articles:

- AVRILOS First Article (Introduction)
- FPGA/CPLD Design Flow for AVRILOS
- AVRILOS & FPGA

- AVRILOS on SourceForge

History

V1.0 of this Article.

AVRILOS V1.23.

License

About the Author

More than 10 year of Embedded Systems development designing both hardware & software for products, lab prototypes and Automated Testers. Have used numerous micro-controllers/processors, DSP & FPGAs.
More info you can find at my personal site: Ilialex