Introduction to generic programming

Date: 30.09.2014

Author(s): Michal Koziel

Abstract

Generic programming is about writing your code portable, modular and easily maintainable. There are however limits for how much effort you want to put into making your code generic. This article will give an overview of different levels of genericity as well as some guidelines to follow.

What is generic programming

Generic programming is parametrization of code. Code expected to be different for each data type, compiler, operating system or hardware is made abstract while the real values are inserted at compiler or at run time. Code genericity can be done at several levels. At higher levels whole libraries can be swapped in and out or hardware description can be passed during initialization. At lower levels generic programming includes, but is not limited to data types, register access and hardware variant parametrization.

Why generic code?

Embedded programmers work across different compilers and platforms. Anyone who has ported non generic code to a different platform has experienced the frustration when searching for yet another low level statement to be changed. A simple thing such as migrating to a processor with different endianess or memory mapping can cause much trouble if not accounted for from the beginning. A set of few simple macros can save days of work.

Generic code comes in handy when executing your code on a host machine before it gets to run on the real target. All hardware access can be simulated in software. Hardware timers can run as threads, interrupts can be fired any time.

Testing is also made much easier. By using a Unit Testing Framework, code can be tested by applying external input and mocking the running environment while keeping the code under test static. Imagine a simple function to copy files. The function can allow size of a statically allocated copy buffer and data types to be customizable at compile time. By recompiling and retesting the function for a variety of parameter values one can assure a range of operating conditions internally to the function. In system testing whole modules can be compiled in and out to test different implementations. You should find that useful in regression testing as well.

Generic programming, if done correctly, also improves quality and readability of the code. This is illustrated in the examples at the end of this article.

Genericity, modularity and architecture

We've all been there at some point, writing a tiny single file program which suddenly starts to grow at an exponential rate. Registers being accessed directly from all parts of the code, addresses and values in hex. Horror! Generic programming is also modular programming and modularity starts with architecture design. Well written software is designed and reviewed before coding starts. Unless you're a programmer with predictability of a chess player you and others will always benefit from planning ahead. Divide your code into layers where processing is done in several steps, as in communication stacks. Define modules for specific and well defined functionality, which applies to everything from high level tasks to lower level infrastructure such as a software FIFO.

How generic should it be?

Making your code too generic can be as bad as not making it generic at all. You should ask yourself "why?", before "what?". Why do you want the code or parts of it generic?

Software such as u-boot and FreeRTOS are written to be highly portable, intended to be run on most platforms. These examples provide complex functionality, but will run on your target after setting only few platform specific parameters. A highly optimized application running on custom hardware is less a subject for generic programming with respect to portability. However, modularity and hardware abstraction should still be an obvious strategy.

Cost of generic code

Writing generic code do come at a cost, but do it right and it will pay back quickly. I can only imagine the effort put into coding and testing of above mentioned u-boot and FreeRTOS and we all appreciate it. On the other hand, spending a lot of your time writing code to run on most platforms might feel as a waste, if it never will . Sometimes you just never know. Make low level generic programming a habit and at least let debugging and maintenance benefit from it. This alone will make it worth the investment.

Examples

Even though most of us practice some sort of hardware access abstraction, bad habits such as the one in the first example below are still frequent.

The second example shows a generic timer module ready to be adapted to any platform.

Example 1

It shouldn't be necessary to lecture that magic numbers should be replaced with constants. Low level register access should also be made abstract. From time to time I read code with hundreds of lines like this one:

```
PORTB = PORTB | (1 << 4);
```

Sure you can see that bit 4 for PORTB is set, but what happens when the bit is set? What was the programmers intention? Was he turning something on or off and what if the polarity changes in hardware? Here is one way to solve this problem:

First you define a macro to set a bit and you throw in a macro for clearing a bit as well.

```
#define SET_BIT(reg, bit) (reg |= (1 << bit))

#define CLEAR_BIT(reg, bit)   (reg &= ~(1 << bit))
```

We got some portability, but readability is still as bad as it was since the below code is still mystical to me.

```
SET_BIT(PORTB, 4);
```

So next step is to assign a logical meaning to this line. Say that pin 4 on PORTB is controlling a LED.

```
#define LED_PORT PORTB
#define LED_BIT 4
#define LED_ON CLEAR_BIT(LED_PORT, LED_BIT);
#define LED_OFF SET_BIT(LED_PORT, LED_BIT);
```

Now, reading, debugging and porting code, where

```
PORTB = PORTB | (1 << 4);
```

is replaced with

```
LED_OFF;
```

makes a whole lot of a difference. Changing port, pin, polarity or turning the the LED on/off is now controlled by 4 simple macros.

Example 2

In this example we will look at a generic timer module which can easily be ported to any platform. This module provides a customizable number of soft timers, where each timer can run in single shot or periodic mode. In addition, an alarm callback function can be assigned and triggered on a timer timeout.

Note, this example is for illustrative purposes. Some code have been left out, such as input parameter checking. Also, the level of genericity might be to high or to low for a specific application.

There are three files, applicationConfig.h, genericTimer.h and genericTimer.c.

applicationConfig.h belongs to the application in which the module is included, while genericTimer.h and genericTimer.c are source files for the timer module.

The applicationConfig.h has three sections. The hardware specific section defines macros for the lowest level code such as setting up registers to start the hardware timer. In a virtual environment the hardware initialization macro can be used to create a thread which calls the timer interrupt service routine periodically. The compiler specific section deals typically with data types and syntax related issues. Finally, the application specific section configures the module and assigns function names to avoid replacing every call from within the application code.

The module header file genericTimer.h should check if all required macros are defined by the application. Good practice is to add a comment explaining the purpose of each macro. The rest of the file defines module specific data types, constants and function prototypes.

genericTimer.c implements all module functionality. The code should be pretty self explanatory. Note, that this module could be further parameterized with respect to the periodic mode and alarm features. The check and call of the alarm functions could be replaced by a single macro. Since this code is run in interrupt context this could be an application defined macro for setting a bit in a shared variable or trigger other interrupts. Another option is to give a choice between a default implementation or a application defined alternative for the alarm handling, still controlled by macros and without any modifications to the module itself.

Table 1 applicationConfig.h

```
/*hardware specific*/
#define GENERIC_TIMER_HW_INIT      /*setup timer registers and start the timer */
#define GENERIC_TIMER_CLEAR_FLAG   /*clear timer interrupt flag*/
#define GENERIC_TIMER_RELOAD       /*left blank for auto reload */
#define GENERAL_TIMER_DISABLE_INT  /*before critical section*/
#define GENERAL_TIMER_ENABLE_INT   /*after critical section*/

/*compiler specific*/
```

```
#define GENERIC_TIMER_TYPE uint32_t
#define GENERIC_TIMER_INTERRUPT_FUNC (interrupt void irq_tmr0(void)) /*interrupt
service routine syntax vary across compilers and platforms*/

/*application specific*/
#define GENERIC_TIMER_COUNT 5
#define GENERIC_TIMER_INIT timerInit
#define GENERIC_TIMER_SET timerSet
#define GENERIC_TIMER_GET timerGet
#define GENERIC_TIMER_SET_ALARM timerSetAlarm
```

Table 2 genericTimer.h

```
#include "applicationConfig.h" /*filename can also be a constant passed with -D
option to the compiler*/

/*for every define used by the module, but defined by the application*/
#ifndef GENERIC_TIMER_HW_INIT
#error "GENERIC_TIMER_HW_INIT is not defined by the application"
#endif


typedef void (*GENERIC_TIMER_ALARM)(void); /*alarm function pointer type*/
#define GENERIC_TIMER_PERIODIC 1
#define GENERIC_TIMER_TIMEOUT(timer) (GENERIC_TIMER_GET(timer) == 0)

void GENERIC_TIMER_INIT(void);
void GENERIC_TIMER_SET (int,GENERIC_TIMER_TYPE, int);
GENERIC_TIMER_TYPE GENERIC_TIMER_GET (int);
void GENERIC_TIMER_SET_ALARM (int, GENERIC_TIMER_ALARM);
```

Table 3 genericTimer.c

```
#include "genericTimer.h"
static GENERIC_TIMER_TYPE timers[GENERIC_TIMER_COUNT];
static GENERIC_TIMER_ALARM timersAlarm[GENERIC_TIMER_COUNT];
static GENERIC_TIMER_TYPE timersPeriode[GENERIC_TIMER_COUNT];

void GENERIC_TIMER_INIT(void)
{
     int i;
     GENERAL_TIMER_DISABLE_INT; /*in case the timer has already been initialized
and is running*/
     for(i = 0; i < GENERIC_TIMER_COUNT; i++)
```

```
	{
		timers[i] = 0;
		timersAlarm[i] = NULL;
		timersPeriode[i] = 0;
	}
	GENERIC_TIMER_HW_INIT;
}
GENERIC_TIMER_INTERRUPT_FUNC
{
	int i;
	GENERIC_TIMER_RELOAD;

	for(i = 0; i < GENERIC_TIMER_COUNT; i++)
	{
		if(timers[i] > 0)
		{
			timers[i]--;
			if(timers[i] == 0)
			{
				timers[i] = timersPeriode[i];
				if(timersAlarm[i] != NULL)
				{
					timersAlarm[i]();
				}
			}
		}
	}
	GENERIC_TIMER_CLEAR_FLAG;
}
void GENERIC_TIMER_SET(int timer, GENERIC_TIMER_TYPE period, int periodic)
{
	GENERAL_TIMER_DISABLE_INT;
	{
		timers[timer] = period;

		if(periodic == GENERIC_TIMER_PERIODIC)
		{
			timersPeriode[timer] = period;
		}
		else
		{
			timersPeriode[timer] = 0;
		}
	}
	GENERAL_TIMER_ENABLE_INT;
}
void GENERIC_TIMER_TYPE GENERIC_TIMER_GET(int timer)
{
	GENERIC_TIMER_TYPE timerValue;

	GENERAL_TIMER_DISABLE_INT;
	{
		timerValue= timers[timer];
	}
	GENERAL_TIMER_ENABLE_INT;
```

```
	return timerValue;
}
void GENERIC_TIMER_SET_ALARM(int timer, GENERIC_TIMER_ALARM alarm)
{
	GENERAL_TIMER_DISABLE_INT
	{
		timersAlarm[timer] = alarm;
	}
	GENERAL_TIMER_ENABLE_INT;
}
```