Don’ts and Dos

Last updated 1.5.2017

Don’t

```
GPIOPinWrite (GPIO_PORTG_BASE, GPIO_PIN_3 | GPIO_PIN_2, 12);
```

Do

```
GPIOPinWrite (GPIO_PORTG_BASE, GPIO_PIN_3 | GPIO_PIN_2,
                GPIO_PIN_3 | GPIO_PIN_2);
```

to set the pins.

Don’t

```
If (GPIOPinRead (GPIO_PORTG_BASE, GPIO_PIN_7) == 128)
```

Do

```
If (GPIOPinRead (GPIO_PORTG_BASE, GPIO_PIN_7) == GPIO_PIN_7)
```

to check if the pin is set.

Don’ts and Dos

Don’t

ignore the fact that any counter which indicates elapsed time will eventually overflow.

Do

build in features which allow a simple counter to reliably provide intervals and analyse the algorithm to ensure it is reliable.

Don’t

read the GPIO port values several times within your interrupt service routine, since the port inputs could change between reads.

Do

read the GPIO port pin values with a single atomic action and store the result in a local variable.

Don’ts and Dos

Don’t

assume that a variable that is declared as **`uint64_t`** or **`long long`** will be operated on atomically.

Do

treat a variable that is declared as **`uint64_t`** or **`long long`** just as you would a pair of 32-bit numbers.

Don’t

try to get a low PWM cycle rate (lower than about 160 Hz with a CPU clock rate of 20 MHz) without adjusting the PWM clock.

Do

see PWM generation lecture. The PWM counter is only 16 bit. You need to call:

```
SysCtlPWMClockSet(SYSCTL_PWMDIV_2);
```

or similar to set a lower (in this example lower by a factor of 2) PWM clock.

Don’ts and Dos

Don’t

cut and paste code from another person’s program without updating the comments.

Do

add new comments for each section of code that are meaningful and correct to indicate you know what you are doing and to remind you why a particular feature is the way it is.

Don’t

write code for any function which is uncommented.

Do

write a brief abstract for each function; this is best done before the code is written and then checked for accuracy once the code has been tested.

Don’ts and Dos

Don’t

call display functions from within an interrupt service routine

Do

set a debugger breakpoint within an interrupt service routine to confirm that an interrupt is being serviced.

Do

set a spare GPIO pin at the start of the ISR and reset it at the end of the ISR; use an oscilloscope to get confirmation that the ISR is being called and to get an estimate of execution time, interrupt timing and/or response time. The extra execution time is of the order of two processor cycles.

Don’ts and Dos

Don’t

clear the GPIO pin change interrupt right at the end of your interrupt service routine (unless you have a specific reason to do so).

Do

clear the GPIO pin change interrupt early in your interrupt service routine, but only after you have performed any reading of the GPIO registers. [The Stellaris documentation comments that the clearing takes several clock cycles to complete.]

Don’t

assume that every peripheral is in a completely reset state when your program commences execution.

Do

before doing any initialisation and configuration of a peripheral, make a call to, e.g.:

```
SysCtlPeripheralReset(SYSCTL_PERIPH_PWM);  // Reset the PWM unit
```

Make sure there is one call per peripheral. *[This was found to be necessary for the Stellaris unit used in previous years. We are unsure if it applies for the Tiva, but the present helicopter demo program has these calls.]*

Don’ts and Dos

Don’t

assume that the order of operations in an expression of integers is unimportant. The compiler performs operations according to brackets, if they are present in the expression, and otherwise evaluates left-to-right. Thus, because of integer arithmetic, i1 * i2 / i3 may evaluate to a different result than i1 / i3 * i2 or (i1 / i3) * i2, with all variables integer.

Do

use integer arithmetic where possible, especially if the final result is going to be converted to an integer anyway, e.g. if the expression is in one of the API calls writing to registers. Integers take less space and fewer cycles to process, in many cases, than floating point values.