EE 3171 Lecture 7

Introduction to TivaWare

Lecture 7 Topics

A General Overview of TivaWare

How to Create Projects using TivaWare

Accessing Registers

Accessing GPIO (General Purpose I/O)

Other TivaWare functionality will be discussed as we talk about the hardware.

Why TivaWare?

TI distributes TivaWare because: *Programming embedded systems is hard.*

TivaWare:

- Simplifies the task a bit for you
- Provides a consistent interface to all peripherals
- Demonstrates what TI considers to be best practices for embedded programming

TivaWare Preliminaries

There are two main folders of important stuff in the TivaWare library:

`driverlib/`

`inc/`

The `driverlib` files are source code for interacting with hardware peripherals.

For the most part, these are written and you don't need to do anything with them.

The `inc` files are just definitions for register names, variables and so on.

These make our job of programming easier.

Accessing Registers

Two Different Methods:

- Direct Register Access (The Traditional Way)
- Software Driver Access (The Easy Way)

Macros are stored in part-specific header files contained in the `inc/` directory; the header file for the TM4C123GH6PM microcontroller is `inc/tm4c123gh6pm.h`).

Including that file makes it:

- Easy to access registers that exist on the device you're using
- Impossible to access registers that don't exist

Direct Register Access

Rules for Direct Register Access are as follows:

Values that end in `_R` are used to access the value of a register.

> `SSI0_CR0_R` is used to access the `CR0` register in the SSI0 module.

All register name macros start with the module name and instance number (for example, `SSI0` for the first SSI module) and are followed by the name of the register as it appears in the data sheet.

> The `CR0` register in the data sheet is `SSI0_CR0_R`.

All register bit fields start with the module name, followed by the register name, and then followed by the bit field name as it appears in the data sheet.

> `SSI_CR0_SPH` is a single bit in the `CR0` register (in the documentation as the `SPH` bit).

It's not at all important that you know what the SPH bit does.

Direct Register Access

Given these definitions, the `CR0` register can be programmed as follows:

```
SSI0_CR0_R = ((5 << SSI_CR0_SCR_S) |
SSI_CR0_SPH | SSI_CR0_SPO | SSI_CR0_FRF_MOTO
| SSI_CR0_DSS_8);
```

Alternatively, the following has the same effect:

```
SSI0_CR0_R = 0x000005c7;
```

TI claims this is not as easy to understand, but this is how I've been programming for ~10 years.

Extracting the value of the SCR field from the `CR0` register is as follows:

```
ulValue = (SSI0_CR0_R & SSI_CR0_SCR_M) >>
SSI0_CR0_SCR_S;
```

More DRA

The GPIO modules have many registers that do not have bit field definitions.

For these registers, the register bits represent the individual GPIO pins; so bit zero in these registers corresponds to the `Px0` pin on the part (where *x* is replaced by a GPIO module letter), bit one corresponds to the `Px1` pin, and so on.

DRA Pros and Cons

Pros:

Fast, lean code.

Efficient code.

Cons:

Requires in-depth knowledge of every hardware system (and its little nuances).

Can be downright confusing.

Software Driver Access

TI's preferred method of accessing the hardware.

Register names are hidden, many calculations are done for you and often a single function call will replace multiple DRA statements.

```
SSIConfigSetExpClk(SSI0_BASE, 50000000,
SSI_FRF_MOTO_MODE_3, SSI_MODE_MASTER,
1000000, 8);
```

Does exactly* the same thing as all the writes three slides ago.

* Could calculate a slightly different value for *one* field.

Not Mutually Exclusive

Using these two approaches is not an “either/or” proposition.

Consider:

Configuring a peripheral is not performance-critical. It’s *operation* is.

Therefore, use SDA for configuration and DRA for routine operation.

Some peripherals are just slow (e.g., serial communications).

Use SDA for those all the time.

SDA Pros and Cons

Pros:

Much easier to understand

Abstracts some confusing details away.

Cons:

Code may not be efficient or minimal.

Might let you do too much without understanding what you're doing.

Simple Example

Using the TivaWare libraries to make the LED blink.

The Easy Way

TI provides a project we can copy and modify for our own purposes with all of the hooks done for you.

SysCtlPeripheralEnable()

```
void SysCtlPeripheralEnable(uint32_t
ui32Peripheral)
```

Page 263, TivaWare Peripheral Driver Library Guide

This function enables a peripheral. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

`ui32Peripheral` can only be certain values (as defined in the documentation)

SysCtlDelay()

```
void SysCtlDelay(uint32_t ui32Count)
```

`ui32Count` is the number of delay loop iterations to perform.

This function provides a means of generating a constant length delay. It is written in assembly to keep the delay consistent across tool chains, avoiding the need to tune the delay based on the tool chain in use.

The loop takes 3 cycles/loop.

GPIOPinTypeGPIOOutput()

`GPIOPinTypeGPIOOutput(uint32_t ui32Port, uint8_t ui8Pins)`

Configures pin(s) for use as GPIO outputs.

`ui32Port` is the base address of the GPIO port.
`ui8Pins` is the special constant describing which pin(s).

The GPIO pins must be properly configured in order to function correctly as GPIO outputs. This function provides the proper configuration for those pin(s).

GPIOPinWrite()

Writes a value to the specified pin(s).

```
void GPIOPinWrite(uint32_t ui32Port,
uint8_t ui8Pins, uint8_t ui8Val)
```

`ui32Port` is the base address of the GPIO port.
`ui8Pins` is the special representation of the pin(s).
`ui8Val` is the value to write to the pin(s).

Writes the corresponding bit values to the output pin(s) specified by `ui8Pins`. Writing to a pin configured as an input pin has no effect.

TivaWare

The TivaWare libraries are extensive for all of the I/O devices, which we'll talk about when we talk about each device.