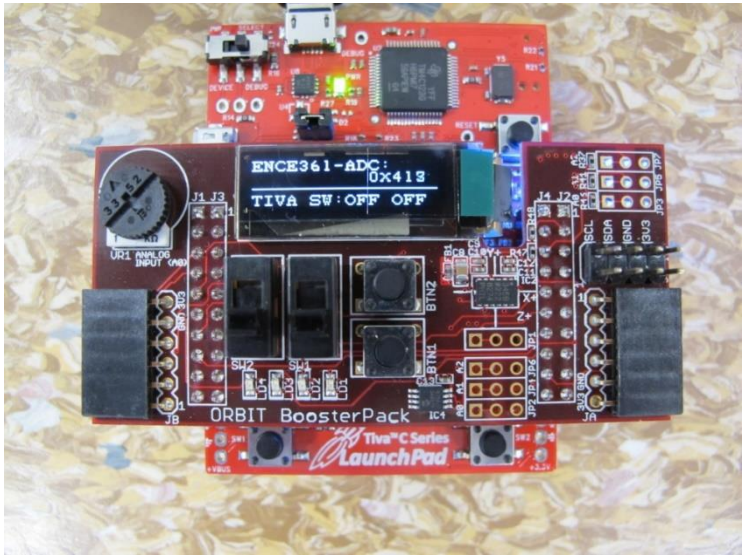
ENCE361 Embedded Systems 1

Interrupts



7th March 2018

Phil Bones

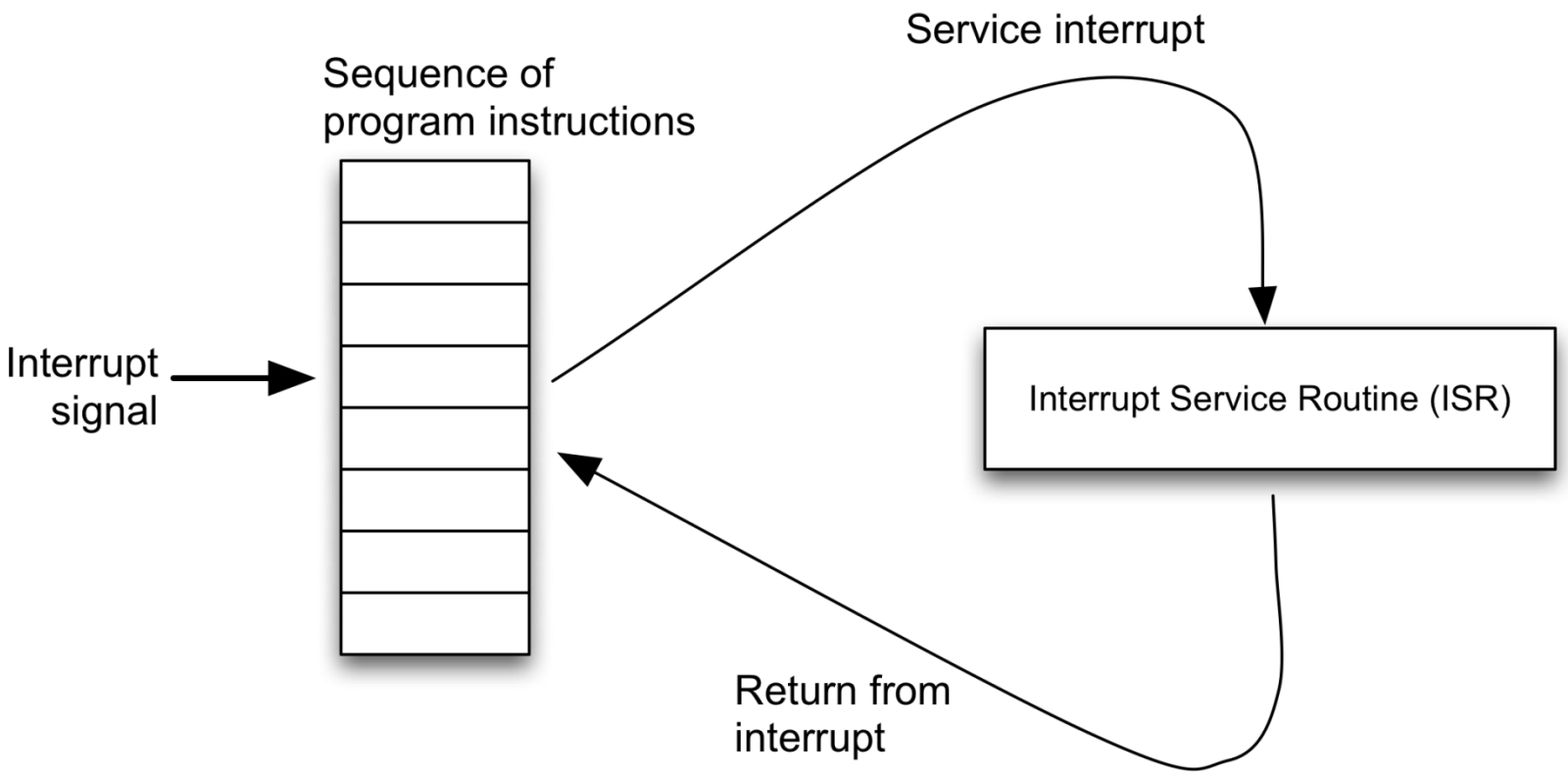

Woof!
Woof!
Woof!
shutterstock
"Someone still has his cellphone on."
www.shutterstock.com · 267029945

Some terminology

- An *interrupt* is an event which causes normal execution of a program to be suspended briefly while a small section of higher priority code is executed. Most microcomputers have a number of interrupt types.
- An *interrupt service routine* (ISR)* is a routine (e.g., a function in C) which is specifically designed to act upon a particular event which generates an interrupt.
- An *interrupt vector* is the address, usually stored in a specific table (the vector table) associated with the ISR for a specific interrupt.
- The *priority* of an interrupt or of a task represents the relative importance of that interrupt or task. Some interrupts have fixed priorities fixed in hardware, others may be programmed.
- An interrupt is *enabled* if the hardware or firmware controlling it is set to respond; it is otherwise *disabled*. If an interrupt is disabled when the event triggering it occurs, it may respond as soon as it is re-enabled.

* The term interrupt "handler" is also applied to an ISR.

Anatomy of an interrupt - basic

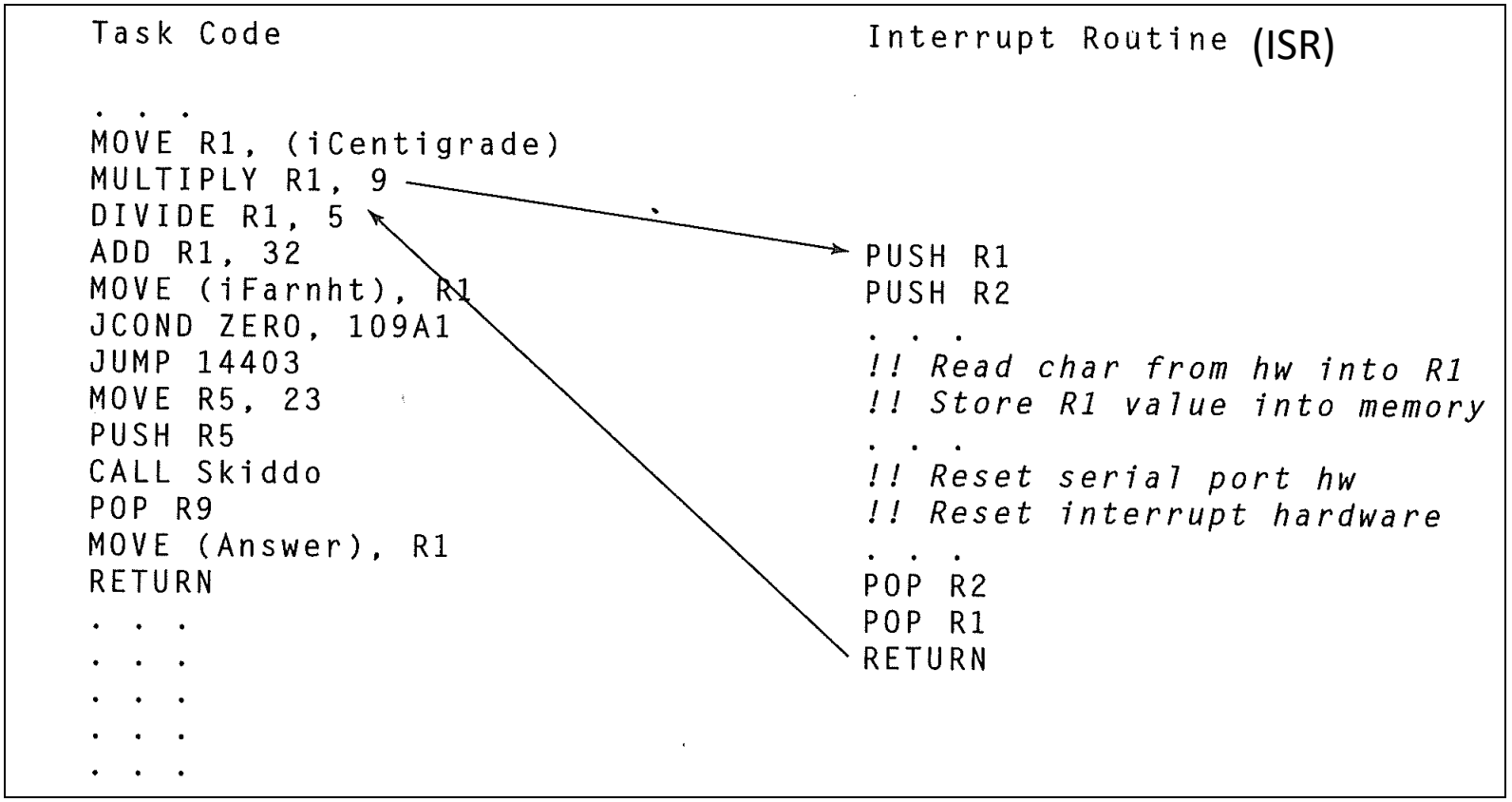


Servicing ("handling") an interrupt

1. The currently executing instruction is completed.
2. Most interrupts *may be* globally disabled.
3. Some part of the context is automatically stored, typically on the stack (the program counter at least).
4. The interrupt vector is loaded into the program counter.
5. The ISR executes, terminating with a return from interrupt instruction.
6. The context, which was automatically stored on initiating the interrupt, is automatically restored from the stack.
7. Most interrupts *may be* globally re-enabled.
8. Execution of the interrupted code continues.

Context

An assembly language example (for an unknown processor) helps to show why the context needs to be saved. The task code is to convert temperature units and uses R1. If the interrupt occurs at the point shown and the ISR uses R1 . . . .



Context

The Cortex-M4 has a stack based *exception* model. When an exception (interrupt) takes place, the Program Counter, Program Status Register, Link Register and the R0-R3, R12 general purpose registers are automatically pushed onto the stack. The data bus is used to stack the registers whilst the instruction bus identifies the exception vector from the vector table and fetches the first instruction of the ISR.

Once the stacking and instruction fetch are completed, the ISR is executed.

After the interrupt service routine has finished, the registers are automatically restored to enable the interrupted program to resume normal execution.

By handling the stack operations in hardware, the Cortex-M4 processor removes the need to write assembler wrappers that are required to perform stack manipulation for traditional C-based interrupt service routines, *unless there is additional context that needs to be preserved*.

Interrupt vectors

Table 2-8. Exception Types

Exception Type	Vector Number	Priority[a]	Vector Address or Offset[b]	Activation
-	0	-	0x0000.0000	Stack top is loaded from the first entry of the vector table on reset.
Reset	1	-3 (highest)	0x0000.0004	Asynchronous
Non-Maskable Interrupt (NMI)	2	-2	0x0000.0008	Asynchronous
Hard Fault	3	-1	0x0000.000C	-
Memory Management	4	programmable[c]	0x0000.0010	Synchronous
Bus Fault	5	programmable[c]	0x0000.0014	Synchronous when precise and asynchronous when imprecise

Table 2-9. Interrupts

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
0-15	-	0x0000.0000 - 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C

. . . continued on following pages of the datasheet

Tiva processor datasheet - **`tm4c123gh6pm.pdf`**

Anatomy of an ISR - detail

Prologue:

- Save the context on the stack (the part not automatically saved, but possibly affected by the servicing).
- If pre-emption by higher priority interrupts is allowed, globally enable interrupts.

Service:

- Perform the essential task related to the interrupt, e.g. copy an ADC output value to a buffer in memory and update counters, flags, etc.
- If necessary, reset the state of the interrupting hardware.
- (If pre-emption by higher priority interrupts is allowed, globally disable interrupts.)

Epilogue:

- Restore the part of the context saved on entry
- Execute a return from interrupt.

Example: extracts from Phil's ADCdemo1.c

```
//
// ADCdemo1.c - Simple interrupt driven program which samples with AIN0
//
. . . .

// Constants
//*****************************************************************
#define BUF_SIZE 10
#define SAMPLE_RATE_HZ 10

// Global variables
//*****************************************************************
static circBuf_t g_inBuffer;// uint32_t buffer of size BUF_SIZE (intervals)
volatile static uint32_t g_u32IntCnt;// Counter for interrupts
. . . .

//*****************************************************************
// The interrupt handler for the for the SysTick interrupt.
//*****************************************************************
void
SysTickIntHandler(void)
{
    // Initiate a conversion
    ADCProcessorTrigger(ADC0_BASE, 3);
    g_ulSampCnt++;
}
```

Example: Phil's ADCdemo1.c, cont.

```
// ***************************************************************
//
// The handler for the ADC conversion complete interrupt.
// Writes to the circular buffer.
//
//***************************************************************
void
ADCIntHandler(void)
{
        uint32_t ulValue;

        //
        // Get the single sample from ADC0.  ADC_BASE is defined in
        // inc/hw_memmap.h
        ADCSequenceDataGet(ADC0_BASE, 3, &ulValue);
        //
        // Place it in the circular buffer (advancing write index)
        writeCircBuf (&g_inBuffer, ulValue);
        //
        // Clean up, clearing the interrupt
        ADCIntClear(ADC0_BASE, 3);
}
```

Example: Phil's ADCdemo1.c, cont.

```
//*****************************************************************************
// Initialisation functions for the clock (incl. SysTick), ADC, display
//*****************************************************************************
void
initClock (void)
{
    // Set the clock rate to 20 MHz
    SysCtlClockSet (SYSCTL_SYSDIV_10 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                    SYSCTL_XTAL_16MHZ);
    //
    // Set up the period for the SysTick timer.  The SysTick timer period is
    // set as a function of the system clock.
    SysTickPeriodSet(SysCtlClockGet() / SAMPLE_RATE_HZ);
    //
    // Register the interrupt handler
    SysTickIntRegister(SysTickIntHandler);
    //
    // Enable interrupt and device
    SysTickIntEnable();
    SysTickEnable();
}
```

Example: Phil's ADCdemo1.c, cont.

```
void
initADC (void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);
    ADCSequenceConfigure(ADC0_BASE, 3, ADC_TRIGGER_PROCESSOR, 0);
    ADCSequenceStepConfigure(ADC0_BASE, 3, 0, ADC_CTL_CH0 | ADC_CTL_IE |
                             ADC_CTL_END);
    ADCSequenceEnable(ADC0_BASE, 3);
    ADCIntRegister (ADC0_BASE, 3, ADCIntHandler);
    ADCIntEnable(ADC0_BASE, 3);
}

int main(void)
{
    initClock ();
    initADC ();
    initDisplay ();
    initCircBuf (&g_inBuffer, BUF_SIZE);

    IntMasterEnable(); // Enable interrupts to the processor.

    while (1)
    {
        // Background task:
        . . . .
    }
}
```

Putting the ISR address in the Vector Table - Method 1

Interrupts can be configured at run-time using calls to the interrupt controller API* function or to a specific API associated with one of the peripherals:

```
IntRegister ()
```

When using **`IntRegister ()`**, the interrupt must also be enabled.

The call required at runtime for setting up a handler for SysTick is:

```
IntRegister (FAULT_SYSTICK, SysTickIntHandler);
```

Equivalently (as in the example code on Slide 12):

```
SysTickIntRegister (SysTickIntHandler);
```

`FAULT_SYSTICK` is defined in the header file **`hw_ints.h`** and represents the offset within the interrupt vector table associated with SysTick.

*The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, to register interrupt ISRs and to set the priority of interrupts.

*The interrupt controller API provides a set of functions for dealing with the Nested Vectored Interrupt Controller (NVIC). Functions are provided to enable and disable interrupts, to register interrupt handlers and to set the priority of interrupts.

```
void IntDisable (uint32_t ui32Interrupt)
void IntEnable (uint32_t ui32Interrupt)
uint32_t IntIsEnabled (uint32_t ui32Interrupt)
bool IntMasterDisable (void)
bool IntMasterEnable (void)
void IntPendClear (uint32_t ui32Interrupt)
void IntPendSet (uint32_t ui32Interrupt)
int32_t IntPriorityGet (uint32_t ui32Interrupt)
uint32_t IntPriorityGroupingGet (void)
void IntPriorityGroupingSet (uint32_t ui32Bits)
uint32_t IntPriorityMaskGet (void)
void IntPriorityMaskSet (uint32_t ui32PriorityMask)
void IntPrioritySet (uint32_t ui32Interrupt, uint8_t ui8Priority)
void IntRegister (uint32_t ui32Interrupt, void (pfnHandler)(void))
void IntTrigger (uint32_t ui32Interrupt)
void IntUnregister (uint32_t ui32Interrupt)
```

Note that "Int" here means "interrupt" not "integer".

Putting the ISR address in the Vector Table - Method 2

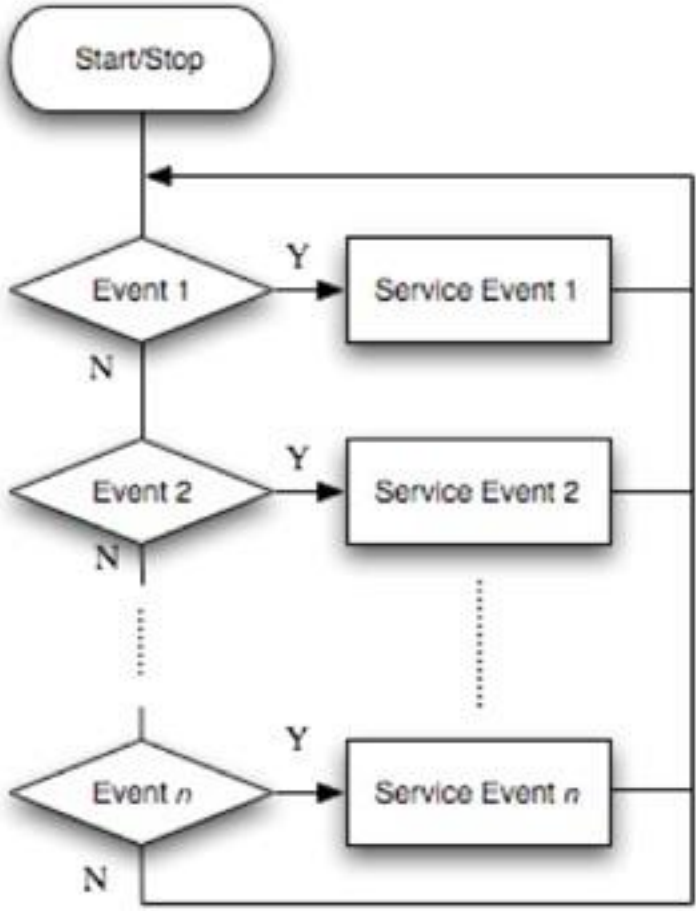
Alternatively, the address of the ISR can be directly coded into the table:

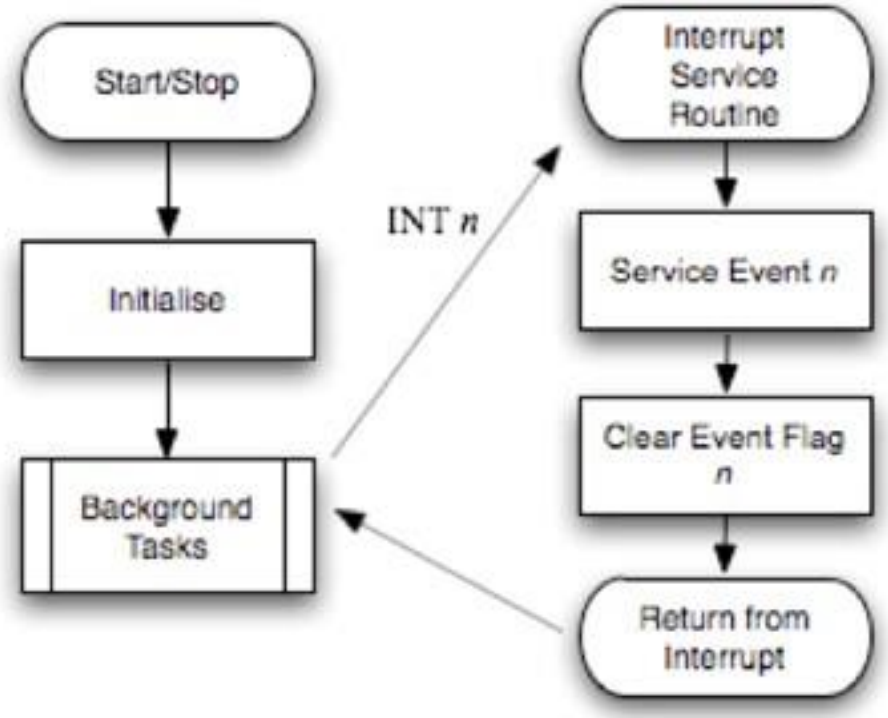
```
//*****************************************************************
// startup_ccs.c - Startup code for use with TI's CCS.
// Copyright (c) 2007-2011 Texas Instruments Incorporated.
//*****************************************************************
// The vector table.  Note that . . .
//*****************************************************************
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
  (void (*)(void))((unsigned long)&__STACK_TOP), // Stack
  ResetISR,                       // The reset handler
  NmiSR,                          // The NMI handler
  FaultISR,                       // The hard fault handler
  IntDefaultHandler,              // The MPU fault handler
            .                               .
            .                               .
  SysTickIntHandler,              // The SysTick ISR
            .                               .
            .                               .
```

Putting the ISR address in the Vector Table

- Which method is best?

Polling vs. Interrupts





When should an interrupt be used?

- If the response to some asynchronous event needs to be completed within a very short period; or
- If the event likely to occur is at a high average repetition rate or at varying intervals, including relatively short ones; or
- If the processor is relatively busy keeping up with the tasks that it must perform; or
- If a foreground/background model simplifies the control code for a real-time system.

References

Simon, D.E. *An Embedded Software Primer*. Addison-Wesley, 1999.

Morton, T.D. *Embedded Microcontrollers*. Prentice-Hall, 2001.

Bennett, S. *Real-time computer control: an introduction*. Prentice-Hall, 1988.

Homework

1. Can Phil's ADCdemo1.c program achieve comparable performance without using interrupts? If so, how? If not, why not?

2. By looking within the processor datasheet **`tm4c123gh6pm.pdf`**, find out how many sorts of interrupts can be generated by the ADC and the Timer modules. List the types (assuming all timers are the same and ADC channels are the same).

3. In general, how can a compiler decide which of the registers need to have their contents saved as part of the context? Refer to Slides 5 and 6.