Networked Embedded Systems

Week 4: Embedded Software

Charalampos (Haris) Orfanidis
Assistant Professor
chaorf@dtu.dk
www.compute.dtu.dk/~chaorf
Slides by Xenofon (Fontas) Fafoutis

Embedded Software

- Embedded Software is software that runs on the processor of an embedded system
- Embedded software is typically closer to the hardware
  - Tied to particular hardware
  - Built around hardware constraints and limitations
  - Controlling sensors, actuators, peripherals, etc
- Also known as firmware
  - Typically, the software that runs on embedded systems does not change as easily or frequently as in general purpose computers
  - It is firmer!
- Most commonly written in C

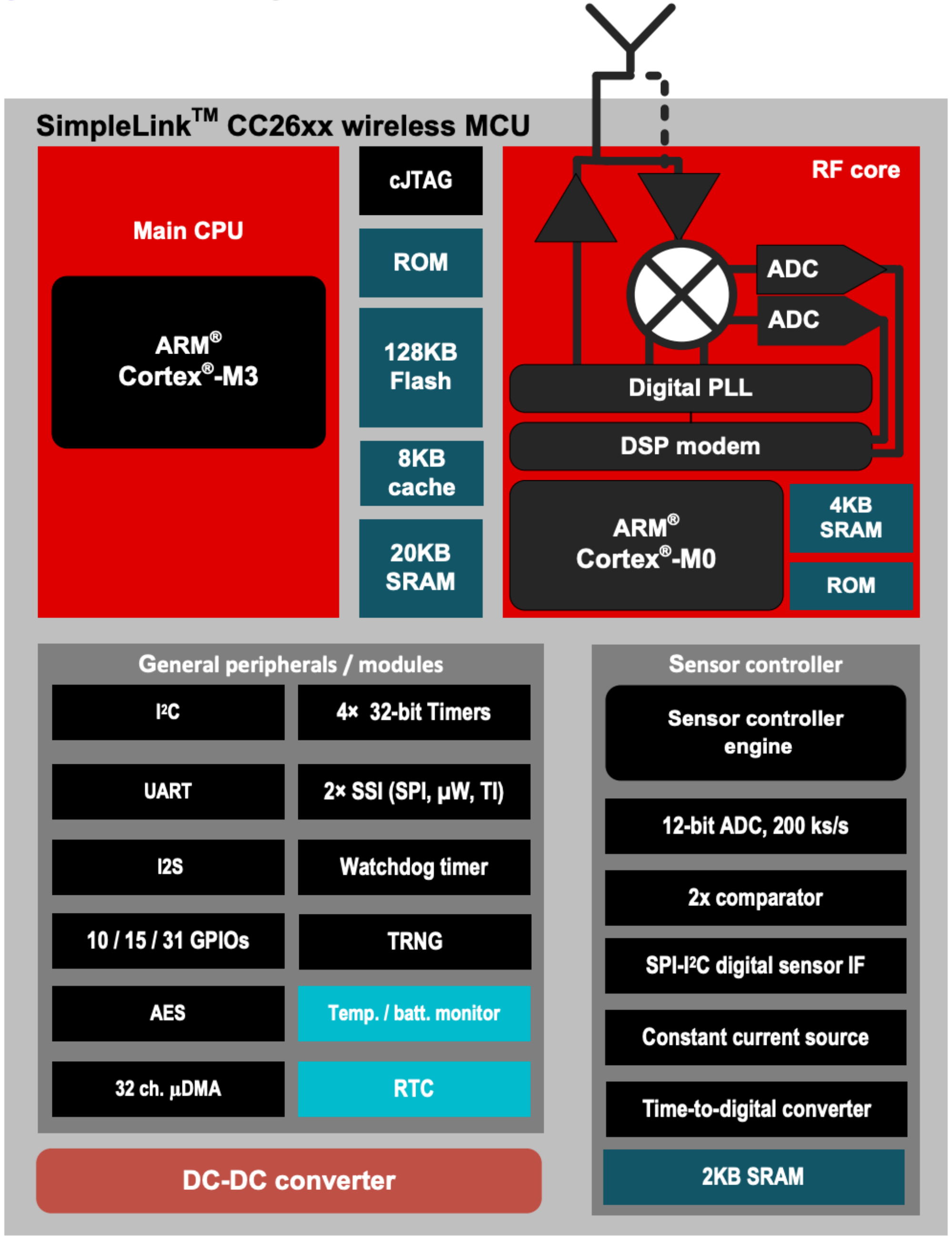
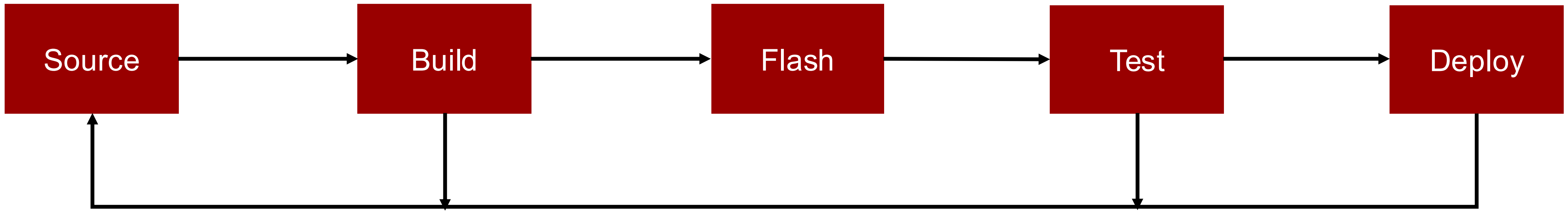


Image source: CC2650 Datasheet by Texas Instruments

Stages of Embedded Software Development

- The development environment is not the same as the execution environment
  - Development takes place at a normal computer
  - Executable binary is installed on the embedded device
    - aka programming or flashing the device
  - Debugging/testing takes place on the embedded device



Embedded Operating Systems

- What is the role of an operating system?

Embedded Operating Systems

- The Operating System (OS) manages and abstracts hardware resources
- However, traditional OS are often not suitable for embedded systems
- Embedded systems without an OS
  - Application talks directly to the hardware (bare metal)
  - Maximum efficiency, but no flexibility
- An Embedded OS provides
  - Hardware abstractions
  - APIs and drivers for common peripherals
  - Intuitive programming models

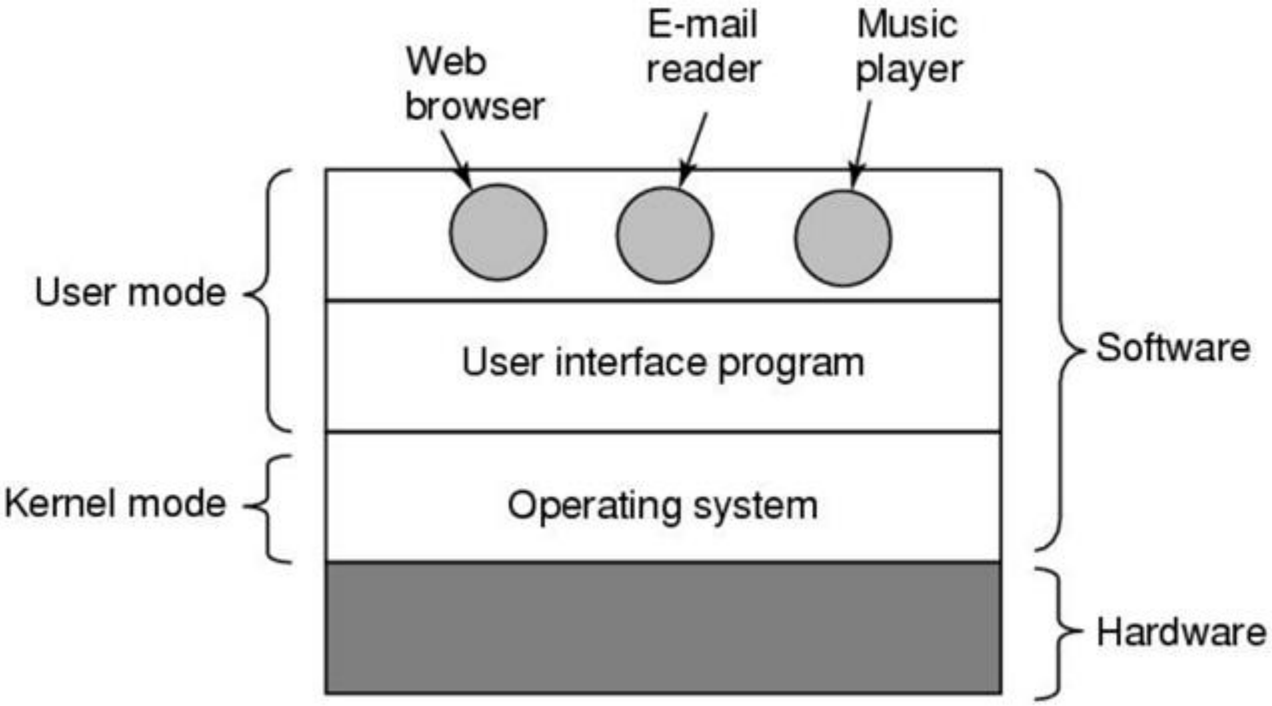
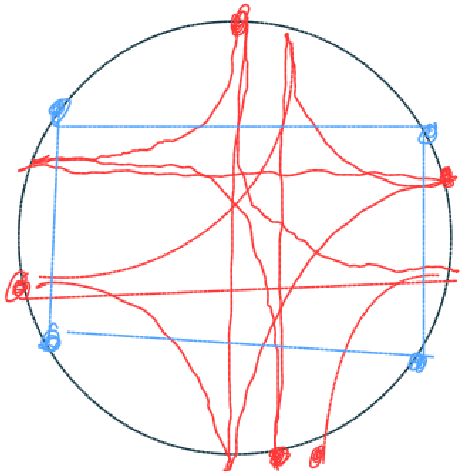


Image source: Modern Operating Systems by Tanenbaum and Bos

Requirements for Embedded OS



- Small memory footprint
  - As low as kbytes in some low-end embedded systems
- Support for heterogeneous hardware
  - Large variety of architectures (8-bit, 16-bit, 32-bit)
  - Variety of available RAM/flash memory
  - Variety of network interfaces (wired, wireless)
- Energy Efficiency
  - OS must duty cycle the CPU/radio/peripherals or provide API to application
- Real-time capabilities
  - Support for timely execution is crucial for various time-sensitive applications
  - An RTOS (Real-Time OS) is an OS that can guarantee worst-case execution times
- Security (confidentiality, authentication, data integrity, access control, etc)

Modules of an Embedded OS

- Embedded OS are designed to be modular
- The application is compiled together with the OS
- The embedded software developer decides which modules to include
  - Through the build system (make)
  - Pre-processor statements (#ifdef)
- The executable binary includes only the absolutely necessary code
- Supporting code for other CPUs, drivers for peripherals not present, protocols not used, etc, are not included in the binary

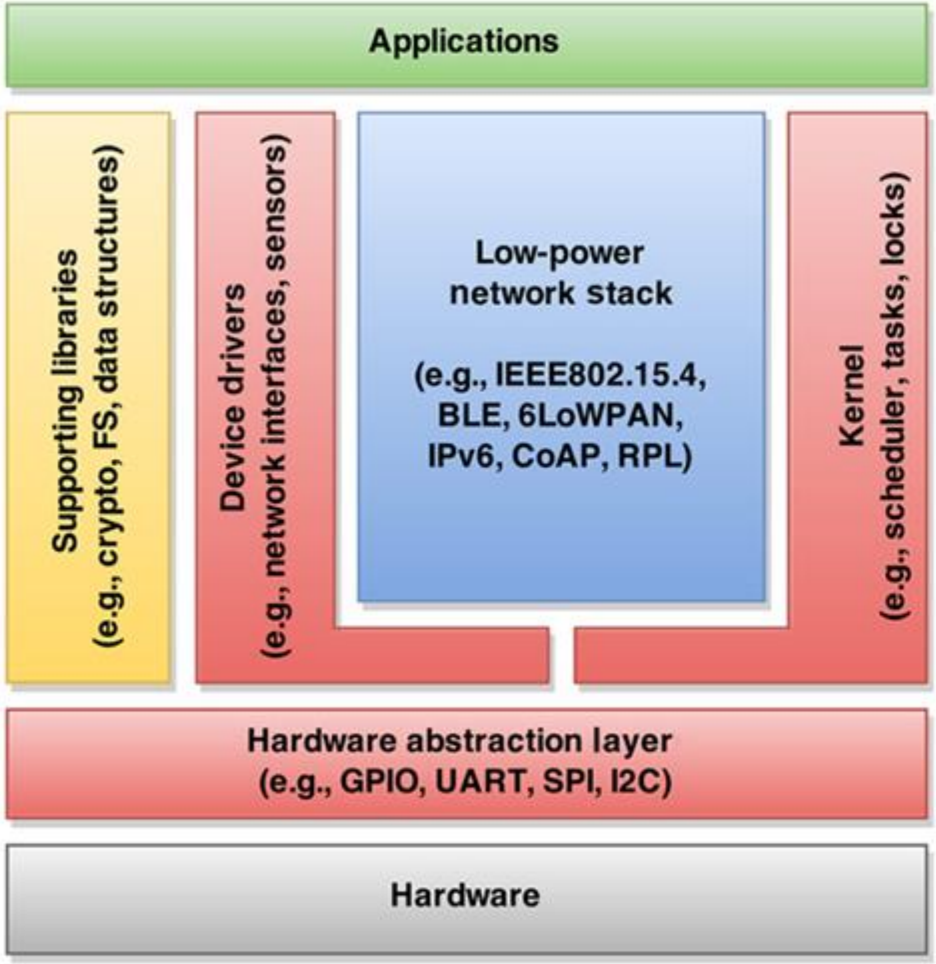


Image source: https://doi.org/10.1109/JIOT.2015.2505901

Event-Driven Embedded OS

- All processing triggered by an external event
  - Events can be generated by peripherals
  - CPU can schedule timer events
- Roughly equivalent to an infinite loop that handles events
- Everything (OS and apps) runs within the same context and address space
  - Like one single programme
- High efficiency in terms of memory and processing
  - Not all programmes can be easily expressed as state machines
- Example: Contiki-NG

Multi-Threading Operating Systems

- Each thread runs in its own context, and manages its own stack
- Scheduler has to perform context switching
- Traditional approach in modern OS (e.g. Linux)
- More natural programming model, but with overhead
  - Memory overhead, runtime overhead
- Example: RIOT

Real-Time Operating Systems (RTOS)

- Primary goal is to provide real-time guarantees
  - In terms of worst-case execution time
- Formal verification, certification, and standardisation
- Strict constraints for the developers
- OS is inflexible and difficult to port to new hardware
- Example: FreeRTOS

Scheduling

- The scheduler manages how the processor is shared among tasks
  - It decides when each task (process or thread) will be executed
- Preemptive Scheduling
  - Scheduler can interrupt the execution of a task to allow another task to use the CPU
  - Fundamental for fairness in UI-based OS
  - Less efficient, more context switches, timer prevents the system from going in sleep
- Non-preemptive Scheduling (cooperative scheduling)
  - Each task executes until it voluntarily yields the CPU
  - More efficient, but vulnerable to errors
  - Tasks must quickly yield the CPU or risk blocking important system functions
- Tickless Preemptive Scheduling
  - Preemption occurs on interrupts and when a task naturally blocks (not time-based)
  - Efficient solution in the middle

Priority Scheduling

- Tasks associated with a priority level
- The scheduler selects the task with the highest priority level
- Highest priority tasks (typically system-level tasks) always get priority over the lower priority tasks
- Lowest priority can suffer from starvation in busy systems
- Rarely the case in embedded systems that spend a lot of time idle/asleep

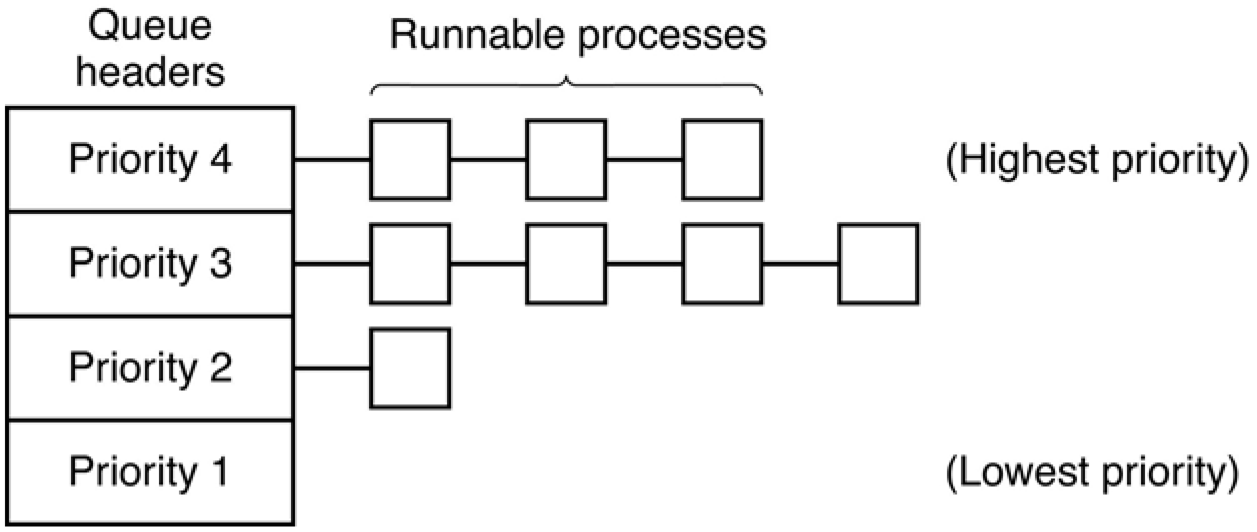


Image source: Modern Operating Systems by Tanenbaum and Bos

Multithreading and Thread Synchronisation

- Parallelism
  - Multicore CPUs
  - Preemptive scheduling enables pseudo-parallel thread execution with one CPU
  - Embedded systems often incorporate auxiliary processors and accelerators (true papalism)

- Parallelism creates the risk of race conditions
  - Shared memory
  - Shared peripherals

- Thread synchronisation tools
  - Mutex
  - Condition variable

- Inter-Process Communication (IPC)
  - Messages

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Image source: Wikipedia

DTU

Embedded Operating Systems

Name	Architecture	Scheduler	Programming model	Targeted device class[a]	Supported MCU families or vendors	Programming languages	License	Network stacks
Contiki	Monolithic	Cooperative	Event-driven, Protothreads	Class 0 + 1	AVR, MSP430, ARM7, ARM Cortex-M, PIC32, 6502	C[b]	BSD	uIP, RIME
RIOT	Microkernel RTOS	Preemptive, tickless	Multithreading	Class 1 + 2	AVR, MSP430, ARM7, ARM Cortex-M, x86	C, C++	LGPLv2	gnrc, OpenWSN, ccn-lite
FreeRTOS	Microkernel RTOS	Preemptive, optional tickless	Multithreading	Class 1 + 2	AVR, MSP430, ARM, x86, 8052, Renesas[c]	C	modified GPL[d]	None
TinyOS	Monolithic	Cooperative	Event-driven	Class 0	AVR, MSP430, px27ax	nesC	BSD	BLIP
OpenWSN	Monolithic	Cooperative[e]	Event-driven	Class 0 – 2	MSP430, ARM Cortex-M	C	BSD	OpenWSN
nuttX	Monolithic or microkernel	Preemptive (priority-based or round robin)	Multithreading	Class 1 + 2	AVR, MSP430, ARM7, ARM9, ARM Cortex-M, MIPS32, x86, 8052, Renesas	C	BSD	native
eCos	Monolithic RTOS	Preemptive	Multithreading	Class 1 + 2	ARM, IA-32, Motorola, MIPS . . .	C	eCos License[f]	lwIP, BSD
uClinux	Monolithic	Preemptive	Multithreading	>Class 2	Motorola, ARM7, ARM Cortex-M, Atari	C	GPLv2	Linux
ChibiOS/RT	Microkernel	Preemptive	Multithreading	Class 1 + 2	AVR, MSP430, ARM Cortex-M	C	Triple License[g]	None
CoOS	Microkernel RTOS	Preemptive	Multithreading	Class 2	ARM Cortex-M	C	BSD	None
nanoRK	Monlothic (resource kernel)	Preemptive	Multithreading	Class 0	AVR, MSP430,	C	Dual License	None
Nut/OS	Monolithic	Cooperative	Multithreading	Class 0 + 1	AVR, ARM	C	BSD	native

Image source: https://doi.org/10.1109/JIOT.2015.2505901

Embedded Operating Systems

OS	Contiki	Contiki-NG	TinyOS	FreeRTOS	OpenWSN	RIOT	Zephyr
MCU	MSP430	MSP430	MSP430	MSP430	MSP430	MSP430	ARM
	AVR	Cortex-M	AVR	AVR	Cortex-M	ARM 7	x86
	Cortex-M	JN516x		Cortex-M		Cortex-M	Xtensa
	ARM 7			Cortex-A		x86	RISC-V
	8051			ARM7		AVR	ARC
	RL78			Cyclone V SOC		ESP8266	Nios II
	6502			ARM9		RISC-V	POSIX/NATIVE
	x86			PIC32			SPARC
				NIOS II			
				8051			
				x86			
				Microblaze			
				APS3			
				78K0R			
				TMS570			
RAM [KB]	10	10	10	4-8	-	1.5	8
Flash [KB]	30	~100	48	32-64	-	5	-
RPL	✓	✓	✓	✗	✓	✓	✓
UDP	✓	✓	✓	✓	✓	✓	✓
TCP	✓	✓	Experimental	✗	✓	✓	✓

Image source: https://doi.org/10.1109/ACCESS.2022.3153521

How do we choose an Embedded OS?

- Features
  - Supported hardware (platforms, CPUs, peripherals, accelerators)
  - Supported software (communication protocols, file system, encryption libraries)
  - Tools (simulator, debugging tools)
- Portability: vendor-based vs generic OS, open standards (POSIX)
- Certification: real-time guarantees, networking interoperability
- License: proprietary, permissive, copyleft
- Documentation, Support, Community

DTU

Embedded Programming

- Typically involves communication with peripherals
  - Internal peripherals in the System-on-Chip (SoC)
  - External peripherals (same board)
- Peripheral controller has a number of registers
  - The CPU reads/writes the registers
  - Types of registers
    - Status registers (read-only)
    - Command registers (write-only)
    - Data/Configuration registers (read/write)
- Bitwise operations, fixed point operations
- Static memory allocation is preferred

Sidder ved

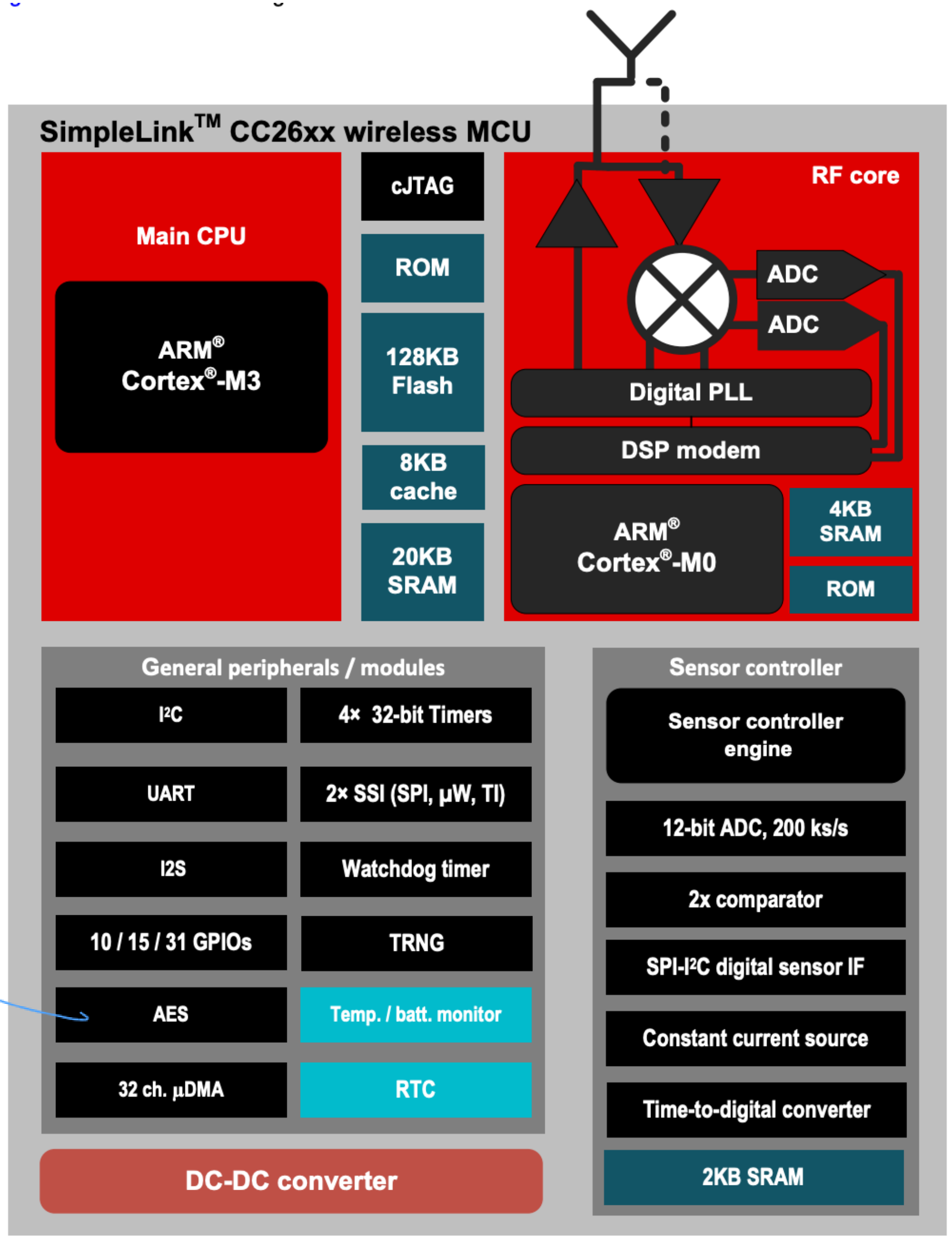


Image source: CC2650 Datasheet by Texas Instruments

Events and Interrupts

- Peripherals can be programmed to issue signals (interrupts) when specific events happen
  - GPIOs can detect changes in line to capture interrupts from external peripherals
  - The interrupt controller handles interrupt for multiple devices

- An Interrupt Request (IRQ) is then sent to the CPU
  - Interrupts normal execution and executes a specified interrupt handler

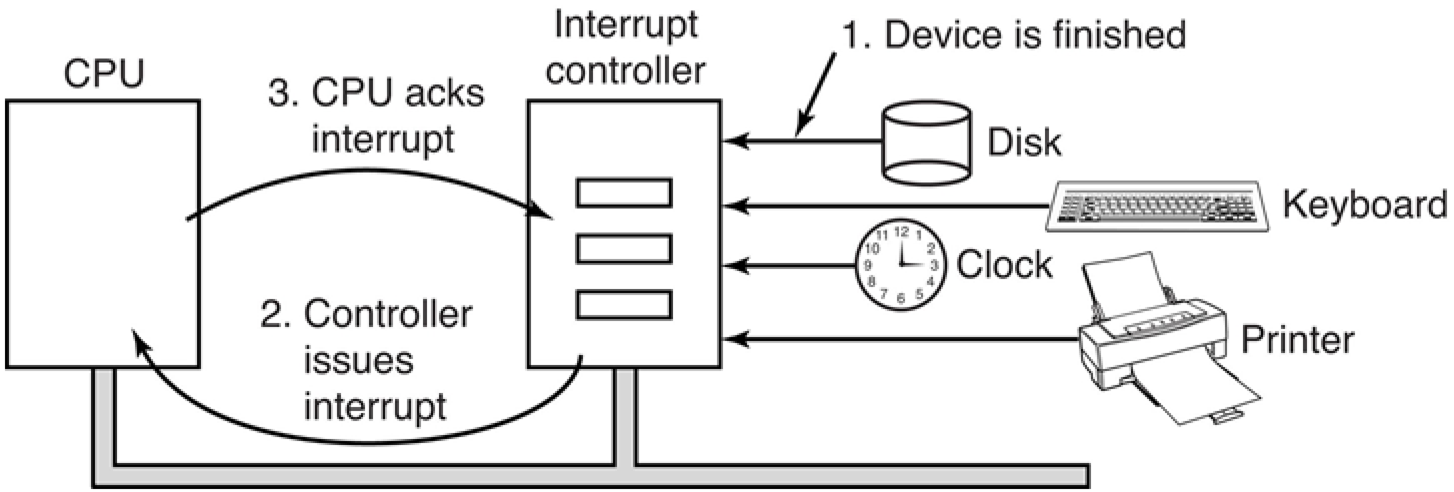
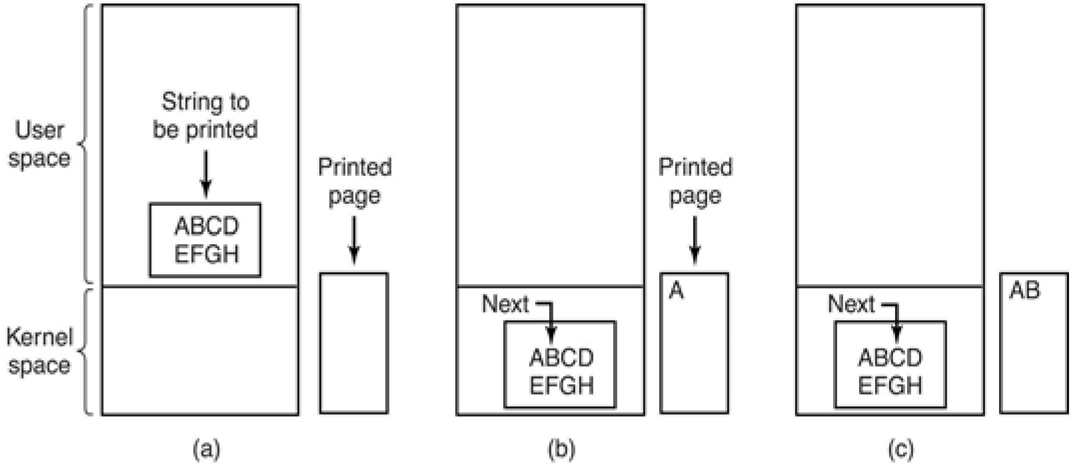


Image source: Modern Operating Systems by Tanenbaum and Bos

I/O Software with Busy Waiting

- Example: Send a string to peripheral (e.g. printer)
  - Copy first character from memory to printer's data register
  - Read printer's status register in a while loop until printer is done
  - Repeat until the end of the document

```
copy_from_user(buffer, p, count);           /* p is the kernel buffer */
for (i = 0; i < count; i++) {               /* loop on every character */
      while (*printer_status_reg != READY) ; /* loop until ready */
      *printer_data_register = p[i];         /* output one character */
}
return_to_user( );
```



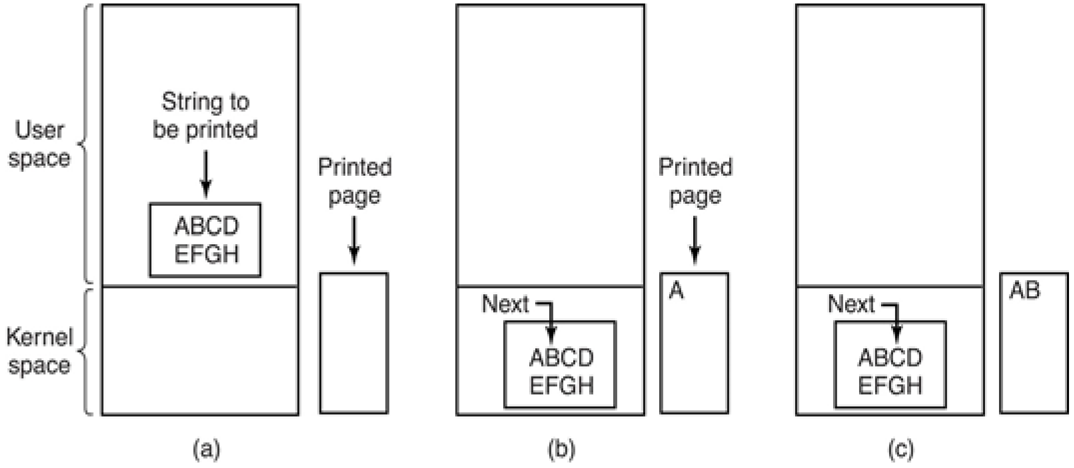
- Disadvantages?

Image source: Modern Operating Systems by Tanenbaum and Bos

I/O Software with Busy Waiting

- Example: Send a string to peripheral (e.g. printer)
  - Copy first character from memory to printer’s data register
  - Read printer’s status register in a while loop until printer is done
  - Repeat until the end of the document

```
copy_from_user(buffer, p, count);                 /* p is the kernel buffer */
for (i = 0; i < count; i++) {                     /* loop on every character */
      while (*printer_status_reg != READY) ;      /* loop until ready */
      *printer_data_register = p[i];              /* output one character */
}
return_to_user( );
```



- Disadvantages?
  - Occupies the CPU doing nothing
  - Wastes energy
  - Heats up the system

Image source: Modern Operating Systems by Tanenbaum and Bos

Interrupt-Driven I/O Software

- Example: Send a string to peripheral (e.g. printer)
  - Enable/configure interrupts
  - Copy first character from memory to printer's data register
  - Put CPU to sleep
  - When peripheral is done it issues an interrupt
  - The interrupt handler wakes up the CPU
  - Copy next character and repeat until the end of file

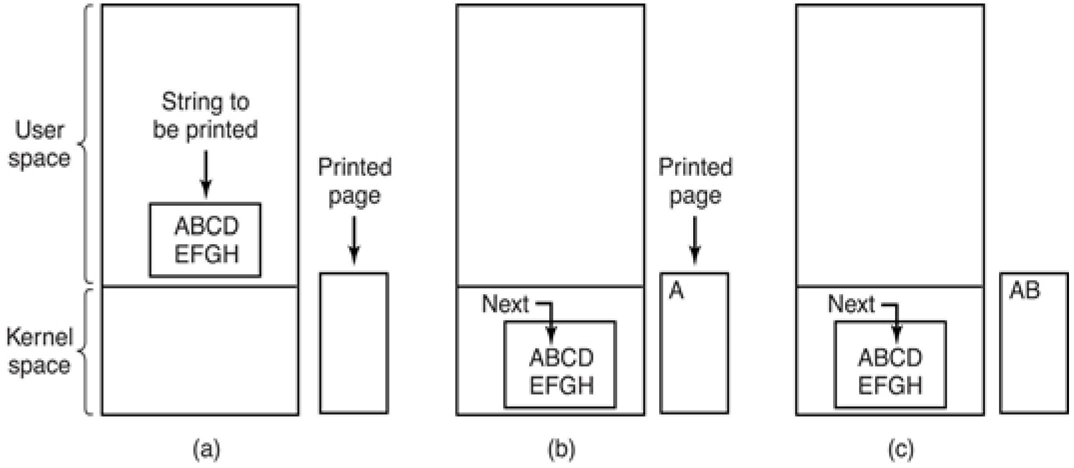
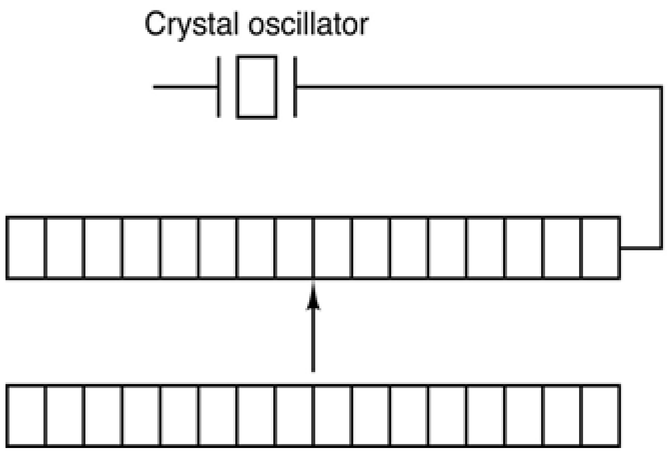


Image source: Modern Operating Systems by Tanenbaum and Bos

Timers



- An SoC would typically have a fixed number of hardware timers
  - High-frequency timers and low-frequency timers (RTC)
  - Example: CC2650 has 2 HF timers and 1 RTC
- A timer has a resolution that depends on its frequency (1/f)
  - A timer with f=32KHz has resolution ~30.5 us
  - A timer with f=24MHz has resolution ~41.7 ns
- A timer will overflow periodically according to its size and frequency
  - A 24-bit timer with f=32KHz will overflow after 512 seconds
  - A 24-bit timer with f=24MHz will overflow after ~0.7 seconds
- Each hardware timer needs to support multiple software timers
  - Timer increments at specific frequency and wraps when reaches maximum value
  - In each timer a COMPARE value can be set, to issue a clock interrupt when timer reaches it
  - The COMPARE value can be updated in software to reset the software timer

Image source: Modern Operating Systems by Tanenbaum and Bos

A Timer Challenge

M K

- Assuming my system has two oscillators 24MHz and 32KHz that drive two 24-bit timers
  - The LF timer with f_{LF}=32KHz will overflow after 512 seconds
  - The HF timer with f_{HF}=24MHz will overflow after ~0.7 seconds

- How do I schedule an event to occur in 5 minutes (300 seconds)?
  - Read the current value of LF timer (COUNTER)
  - Calculate the ticks the correspond to the interval (INTERVAL = 300 x f_{LF} = 9830400)
  - Add ticks to current value, making sure I wrap around timer maximum value
  - COMPARE = (COUNTER + INTERVAL) mod 2^24

- How do I schedule an event to occur in 10 minutes (600 seconds)?

A Timer Challenge

- How do I schedule an event to occur in 10 minutes (600 seconds)?
  - The LF timer with f_{LF}=32768 Hz will overflow after 512 seconds
- Approach #1: Generate more events than needed
  - Schedule an event every 300 seconds, ignore every odd event
- Approach #2: Trade resolution for overflow period
  - Set the PRESCALER=1 to generate a lower timer frequency (f_{LF} = 16384 Hz)
  - Calculate the interval based on reduced frequency (INTERVAL = 600 x f_{LF} = 9830400)
  - Set COMPARE = (COUNTER + INTERVAL) mod 2^24

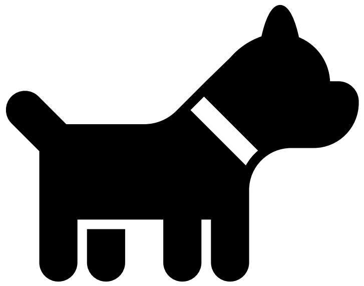
```
fRTC [kHz] = 32.768 / (PRESCALER + 1 )
```

Prescaler	Counter resolution	Overflow
0	30.517 μs	512 seconds
2^8-1	7812.5 μs	131072 seconds
2^{12}-1	125 ms	582.542 hours

Image source: nRF52832 Datasheet by Nordic Semiconductors

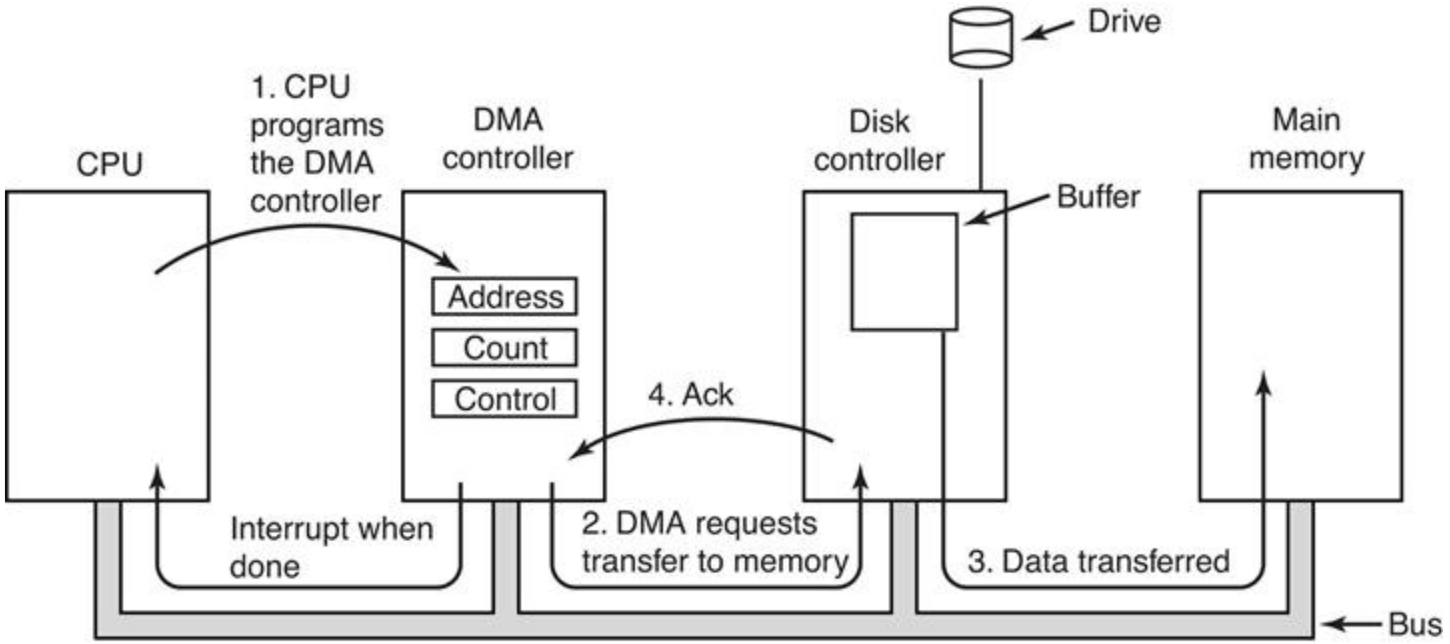
Watchdog Timer

- A safety mechanism against system crashes
- A timer that resets the system when it expires
- Under normal operation the system should reset the timer periodically making sure it never expires
- If the system hangs (deadlock, infinite loop, etc), the watchdog will reset the system
- Can be temporarily paused in long sleep mode or when the system is halted by a debugger



Direct Memory Access (DMA)

- Transfers data between the memory and peripherals without the involvement of CPU
- Support for multiple channels (multiple data transfers)
- Transfer modes:
  - Memory-to-memory
  - Memory-to-peripheral
  - Peripheral-to-memory
  - Peripheral-to-peripheral



- CPU gets notified with an interrupt when transfer done
- Slower but more energy efficient than CPU

Image source: Modern Operating Systems by Tanenbaum and Bos

MCU Power Modes

- MCUs provide granular control on which sub-modules to have active
  - Energy consumption vs functionality
- Active Mode
  - CPU active, RAM on, HF clock on, internal peripherals active if needed
- Idle Mode
  - CPU off, RAM on, HF clock on, internal peripherals active if needed
- Sleep Mode (or Standby Mode)
  - CPU off, RAM retention, HF clock off, internal peripherals unavailable
  - LF clock on, time-scheduled wakeup possible
- Deep Sleep Mode (or Shutdown Mode)
  - CPU off, no RAM retention, HF clock off, LF clock off, internal peripherals unavailable
  - Wakeup on pin edge possible

Example: CC2650 Power Modes

Mode	Software Configurable Power Modes			
	Active	Idle	Standby	Shutdown
System CPU	Active	Off	Off	Off
System SRAM	On	On	Retained	Off
Register retention [(1)]	Full	Full	Partial	No
VIMS_PD (flash)	On	Available	Off	Off
RFCORE_PD (radio)	Available	Available	Off	Off
SERIAL_PD	Available	Available	Off	Off
PERIPH_PD	Available	Available	Off	Off
Sensor controller	Available	Available	Available	Off
Supply system	On	On	Duty-cycled	Off
High-speed clock	XOSC_HF or RCOSC_HF	XOSC_HF or RCOSC_HF	Off	Off
Low-speed clock	XOSC_LF or RCOSC_LF	XOSC_LF or RCOSC_LF	XOSC_LF or RCOSC_LF	Off
Wakeup on RTC	Available	Available	Available	Off
Wakeup on pin edge	Available	Available	Available	Available
Wakeup on reset pin	Available	Available	Available	Available

Image source: CC13xx, CC26xx SimpleLink Wireless MCU Technical Reference Manual by Texas Instruments

Example: CC2650 Consumption

PARAMETER		TEST CONDITIONS	MIN	TYP	MAX	UNIT
I_{core}	Core current consumption	Reset. RESET_N pin asserted or VDDS below Power-on-Reset threshold		100		nA
		Shutdown. No clocks running, no retention		150		nA
		Standby. With RTC, CPU, RAM and (partial) register retention. RCOSC_LF		1		µA
		Standby. With RTC, CPU, RAM and (partial) register retention. XOSC_LF		1.2		µA
		Standby. With Cache, RTC, CPU, RAM and (partial) register retention. RCOSC_LF		2.5		µA
		Standby. With Cache, RTC, CPU, RAM and (partial) register retention. XOSC_LF		2.7		µA
		Idle. Supply Systems and RAM powered.		550		µA
		Active. Core running CoreMark		1.45 mA + 31 µA/MHz		
		Radio RX [1]		5.9		mA
		Radio RX[2]		6.1		mA
		Radio TX, 0-dBm output power[1]		6.1		mA
		Radio TX, 5-dBm output power[2]		9.1		mA

Image source: CC2650 Datasheet by Texas Instruments

Embedded Programming Tricks

- Two's Compliment Numbers for signed numbers
  - To take negative N-bit value, subtract from 2^N
  - Example: in an 8-bit integer, -100 = 256 – 100 = 156 = 0x9C
- Use explicit data types and minimal data types
  - How big is an int? in many embedded systems it is not 32 bits!
  - Use instead: int32_t, uint32_t, int16_t, uint16_t, etc
  - Don't use a uint32_t when a uint8_t is enough
- Static Variables (e.g. static uint8_t A;)
  - Puts variable A in statically allocated piece of global memory (not stack)
  - Variable survives between function calls
- Volatile Variables (e.g. volatile uint8_t A;)
  - Specifies that variable is expected to change by other entities
  - Compiler does not put it in register or cache

Embedded Programming Tricks: Inline Functions

- Inline functions
  - Omit overhead of calling function
  - Speeds up execution of basic functions
  - Copy/pasting code is bad practice for code maintenance

```
inline sum (int a, int b){
  int result;
  result = a + b;
  return result;
}
```

- The two below are identical:
  - `c = sum(a,b);`
  - `c = a + b;`

Embedded Programming Tricks: Macros

- Avoid hardcoded numbers
  - #define TICKS_IN_SECOND 32768
- Remove functionality that you don't need at runtime
  - #ifdef FEATURE … #endif
- Use parenthesis to avoid bugs
  - #define ONE_PLUS_ONE 1+1 -> A = ONE_PLUS_ONE * 10 = 1+1*10 = 11
  - #define ONE_PLUS_ONE (1+1) -> A = ONE_PLUS_ONE * 10 = (1+1)*10 = 20
- Same with function-like macros (use inline functions preferably)
  - #define TIMES_TWO(x) x*2 -> A = TIMES_TWO(1+1) = 1+1*2 = 3
  - #define TIMES_TWO(x) (x)*2 -> A = TIMES_TWO(1+1) = (1+1)*2 = 4

Embedded Programming Tricks: Efficient Math

- Swift left N to multiply by 2^N
  - A = A << 2 is equivalent to A = A * 4

- Swift right N to divide by 2^N (remainder gets lost)
  - A = A >> 2 is equivalent to A = A / 4

- Use fixed point operations instead of floating-point operations
  - Variable T is an int16 that holds temperature in 100ths of ^{o}C
  - T = 2535 means 25.35 ^{o}C
  - T = T >> 1 divides temperature by 2, so T = 1267 that is 12.67 ^{o}C

Specific Bits

Specific Bits

- Often you need to read/set/clear/toggle a specific bit within a byte

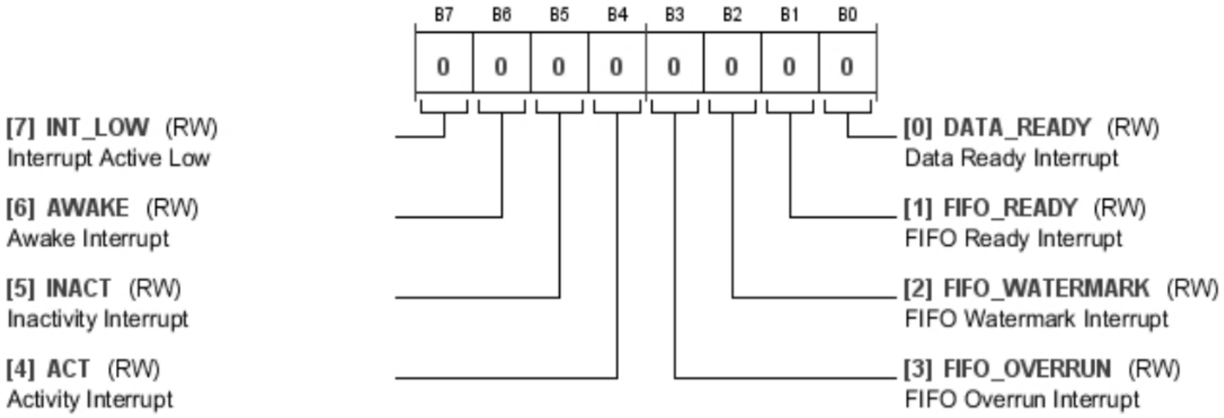


Table 15. Bit Descriptions for INTMAP1

Bits	Bit Name	Settings	Description	Reset	Access
7	INT_LOW		1 = INT1 pin is active low.	0x0	RW
6	AWAKE		1 = maps the awake status to INT1 pin.	0x0	RW
5	INACT		1 = maps the inactivity status to INT1 pin.	0x0	RW
4	ACT		1 = maps the activity status to INT1 pin.	0x0	RW
3	FIFO_OVERRUN		1 = maps the FIFO overrun status to INT1 pin.	0x0	RW
2	FIFO_WATERMARK		1 = maps the FIFO watermark status to INT1 pin.	0x0	RW
1	FIFO_READY		1 = maps the FIFO ready status to INT1 pin.	0x0	RW
0	DATA_READY		1 = maps the data ready status to INT1 pin.	0x0	RW

Image source: ADXL362 Datasheet from Analog Devices

Embedded Programming Tricks: Bitwise Operations

- Often you need to set/clear/toggle/read a specific bit within a byte (uint8_t A)
- Create a mask on a specific bit (left swift moves 0x01 to desired position)
  - M = (0x1<<6); // mask on bit 6, that is 0b01000000
- Setting a bit (logic OR with 1 sets the bit, OR with 0 keeps bit as is)
  - A = A | (0x1<<6); // sets bit 6
- Toggle a bit (logic XOR with 1 inverts the bit, XOR with 0 keeps bit as is)
  - A = A ^ (0x1<<6); // toggles bit 6
- Clear a bit (logic AND with 0 clears the bit, AND with 1 keeps bit as is)
  - A = A & ((0x1<<6) ^ 0xFF); // clear bit 6
- Branch if a bit is set (logic AND with 0 clears the bit, AND with 1 keeps bit as is)
  - if(A & (0x1<<6)){} // branches if bit 6 is set

Sets of Bits

- Often you need to read/write a set of bits within a byte

FILTER CONTROL REGISTER

Address: 0x2C, Reset: 0x13, Name: FILTER_CTL

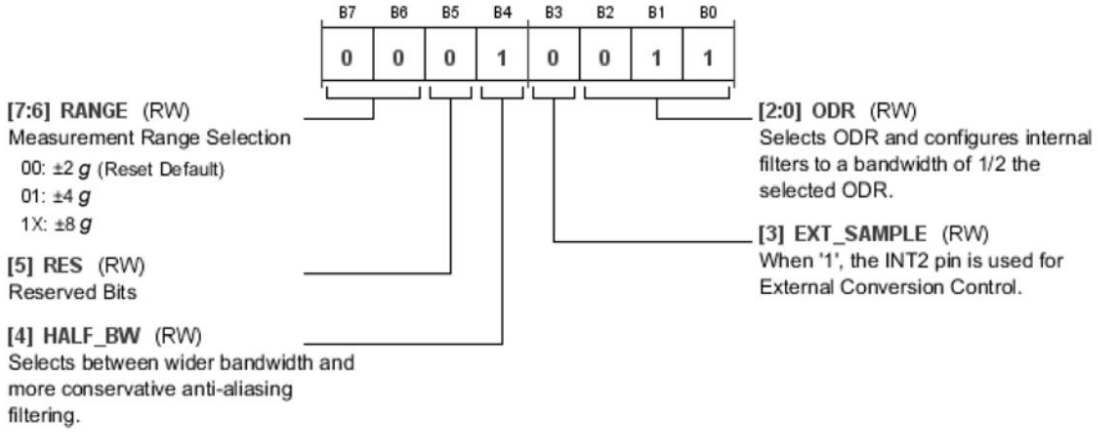


Table 17. Bit Descriptions for FILTER_CTL

Bits	Bit Name	Settings	Description	Reset	Access
[7:6]	RANGE		Measurement Range Selection.	0x0	RW
		00	±2 *g* (reset default)		
		01	±4 *g*		
		1X	±8 *g*		
5	RES		Reserved.	0x0	RW
4	HALF_BW		Halved Bandwidth. Additional information is provided in the Antialiasing section.	0x1	
			1 = the bandwidth of the antialiasing filters is set to ¼ the output data rate (ODR) for more conservative filtering.		
			0 = the bandwidth of the filters is set to ½ the ODR for a wider bandwidth.		
3	EXT_SAMPLE		External Sampling Trigger. 1 = the INT2 pin is used for external conversion timing control. Refer to the Using Synchronized Data Sampling section for more information.	0x0	RW
[2:0]	ODR		Output Data Rate. Selects ODR and configures internal filters to a bandwidth of ½ or ¼ the selected ODR, depending on the HALF_BW bit setting.	0x3	RW
		000	12.5 Hz		
		001	25 Hz		
		010	50 Hz		
		011	100 Hz (reset default)		
		100	200 Hz		
		101…111	400 Hz		

Image source: ADXL362 Datasheet from Analog Devices

Embedded Programming Tricks: More Bitwise Operations

- Often you need to read/write a set of bits within a byte (uint8_t FILTER_CTRL)

- Create a mask on a set of bits
  - MASK3 = (0x1<<3)-1; // mask on first 3 bits, that is 0b00000111
  - MASK2 = (0x1<<2)-1; // mask on first 2 bits, that is 0b00000011

- Read a set of bits
  - ODR = FILTER_CTRL & MASK1; // keeps first 3 bits as is, clears all other
  - RANGE = (FILTER_CTRL>>6) & MASK2; // moves last 2 bits by 6 before masking

- Write a set of bits (without corrupting other bits!)
  - FILTER_CTRL = (FILTER_CTRL & (0xFF ^ MASK3)) | (new_odr & MASK3); // clears the 3 bits first, then OR the new value
  - FILTER_CTRL = (FILTER_CTRL & (0xFF ^ (MASK2<<6))) | ((new_range & MASK2)<<6); // clears the 2 bits first, then OR the new value

Programming Embedded Systems

- Programming or flashing the device
- The building toolchain outputs a binary file that needs to be uploaded on the flash memory of the embedded system
- Done a special programmer/debugger chip/board
  - Receives commands from USB/serial
  - Development boards often come with the programmer/debugger chip on board
- Alternatively, the MCU bootloader may be able to use the UART interface to receive firmware over serial

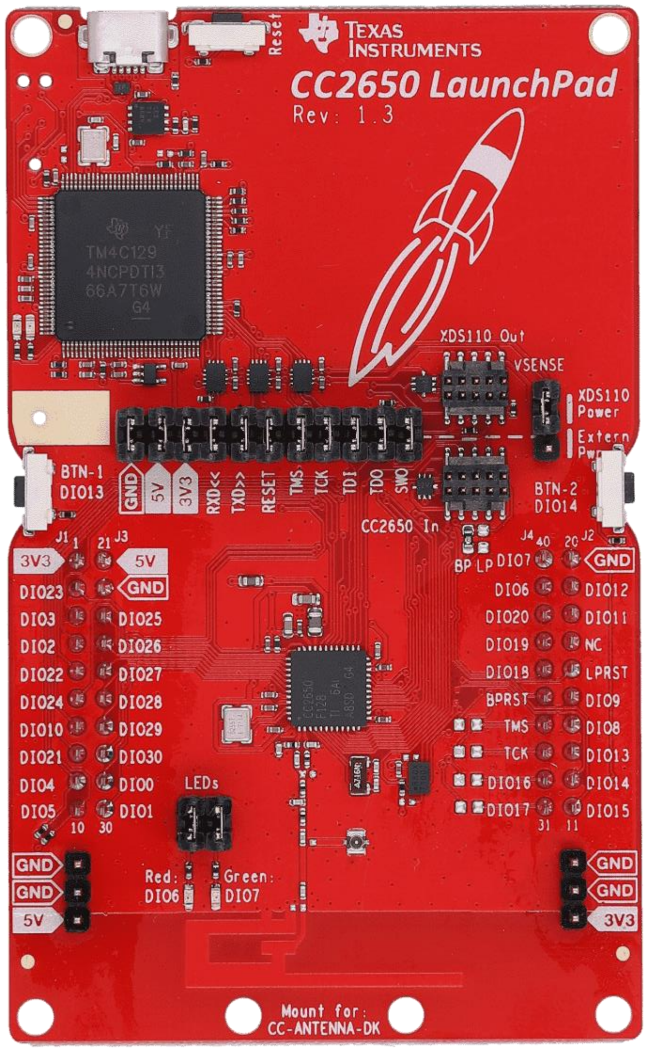
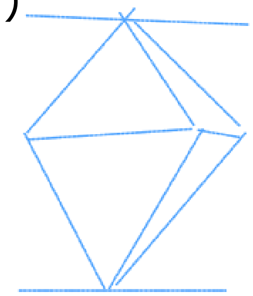
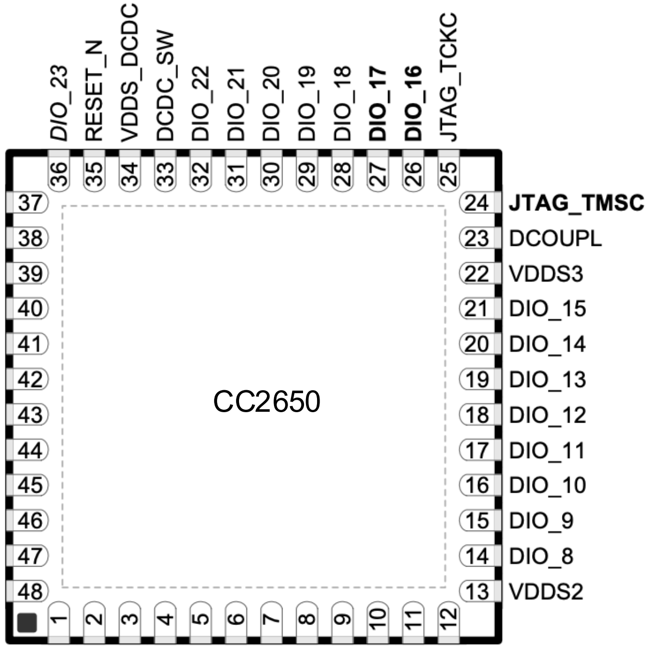


Image source: https://www.ti.com/tool/LAUNCHXL-CC2650

Programming/Debugging Interfaces

- Programming/Debugging Standards
  - Used for uploading binary code and as a debugger
- JTAG (Joint Test Action Group)
  - Generic standard (IEEE 1149.1)
  - 4 wires (TCK, TMS, TDI, TDO) + RESET (optional)
  - Can program multiple devices in a daisy chain
- cJTAG (IEEE 1149.7)
  - 2 wires (TMSC, TCKC) + RESET (optional)
  - Can program multiple devices in a star topology
- SWD (Serial Wire Debug)
  - Specific for ARM processors
  - 2 wires (SWDIO, SWCLK) + RESET (optional)
  - Can program multiple devices in a star topology





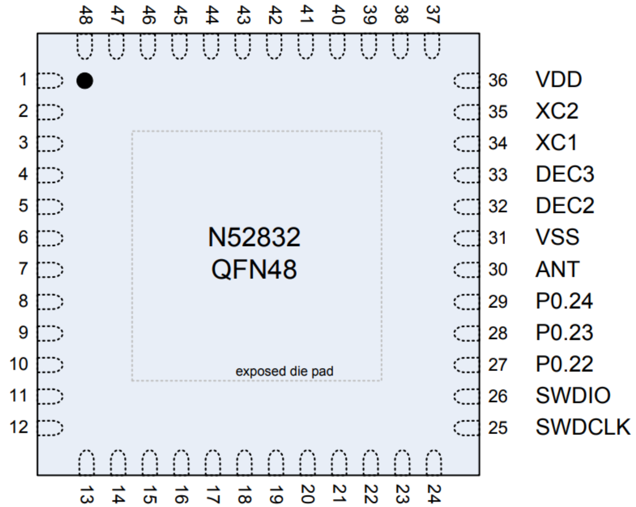


Image source: CC2650 Datasheet by Texas Instruments (top) and nRF52832 Datasheet by Nordic Semiconductors (bottom)

Debugging Embedded Software

- Debuggers are powerful (break points, read memory of embedded system, etc)
  - Break points not compatible with distributed software
  - Not always available in the wild
- LEDs as debugging tools
  - Blink when entering/exiting a code region
  - Turn on when entering an error state
- Printing messages and logs
  - Printing on serial if UART connected to a terminal
  - Printing on a file in flash memory
  - Sending log messages over wireless
- Oscilloscopes, Multi-meters, Logic Analysers
  - Test voltage levels and logic levels

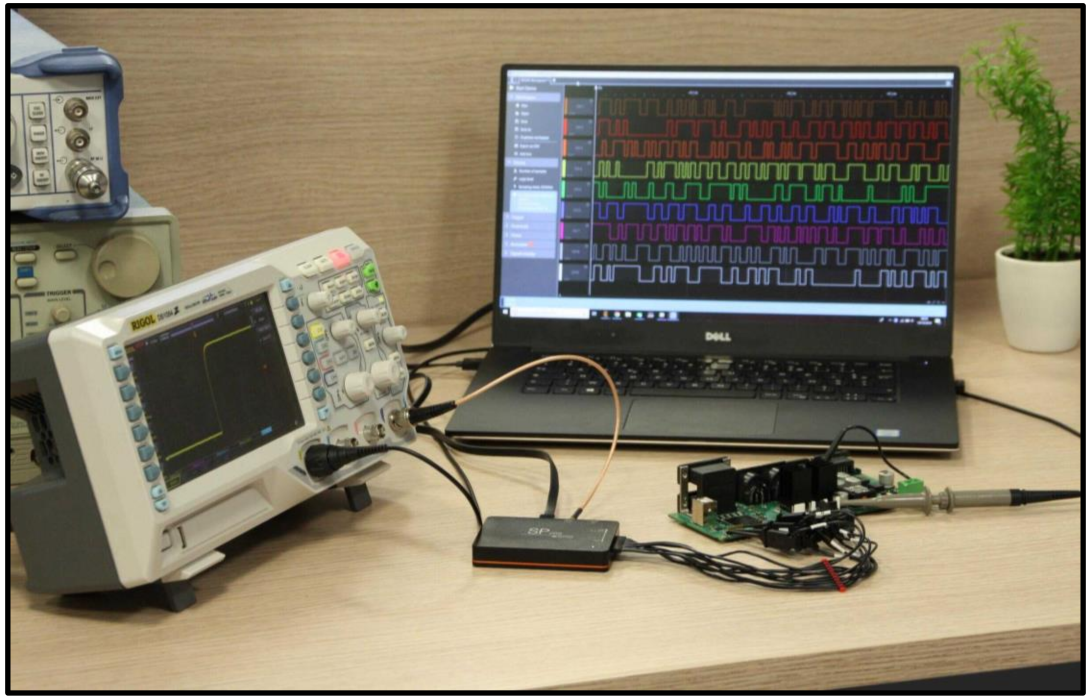


Image source: https://www.ikalogic.com/sp209-logic-analyzer/