Embedded Systems

Software in Embedded Systems

George Kornaros

Contents

- embedded software requirements
- support for developers, an overview
  - ✓standalone applications
  - ✓operating systems and kernels
  - ✓virtualization
  - ✓drivers

Embedded Software



- very simple
  (e.g. temperature sensor: polling, single processor, single thread, few lines of assembly code)
- very complex
  (e.g. mobile phone: multiple processor, multi-threaded, powerful operating systems, thousands of lines of code, legacy libraries)

- Large variation in complexity !!!

Embedded Software



- highly critical
  (e.g. fly-by-wire: hard real-time, safety critical, redundant, certification, offline modeling and testing)
- mostly harmless
  (e.g. Gameboy: quality of service oriented, replaceable, failure is annoying at most)

- Large variation in Requirements!!!

Overview

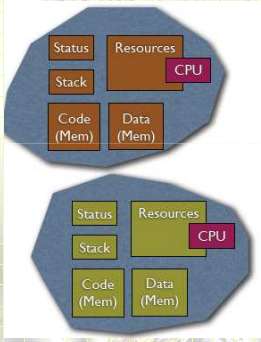
- Embedded Systems usually must:
  - work with the environment
    (monitor, process, control)
  - keep some sort of timing
    (deadlines, QoS, control quality)
  - use limited resources, meaning...
    (power, processing, bandwidth, memory)
  - ...have a low cost
    (development, fabrication, maintenance,...)

Embedded Software Requirements

- Embedded software:
  1. execute concurrent activities
     (multiprogramming, resource sharing)
  2. handle various I/O at different levels
     (devices, drivers, polling or interrupts)
  3. detect faults, fail gracefully (exceptions)
  4. support timing (RT, deadlines, priorities)
  5. easy to develop, verify, manage... (tools)

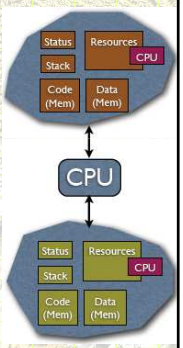
Concurrency

- Processes (Tasks)
  - ✓Running programs
  - ✓Separate code and data
  - ✓Shared resources
  - ✓Appear as executing simultaneously
  - ✓Can communicate



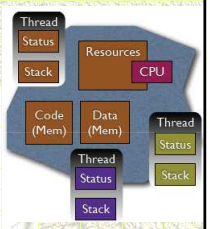
Concurrency - contexts

- executing (scheduling) tasks on one processor
  - ✓multi-programming (events)
  - ✓time sharing (timer)
  - ✓cooperative multitasking (yield control)
  - ✓real-time (fixed points)
- context switch
  - ✓state, registers (program counter, stack pointer/frame), held resources, etc.



Concurrency - Threads

- Threads
  - ✓smallest subset of resources needed for Execution
  - ✓common code/data
  - ✓many shared resources
  - ✓fast context switch within a process



Concurrency - Types

a. single process single thread - SPST (very simple applications)

b. multi-process single thread - MPST

c. single process multi-thread - SPMT (may already reside in memory)

d. multi-process multi-thread - MPMT (desktop PC, Windows, Linux,...)

SPST

- usually a single infinite loop
  (no scheduling, as such)
- use polling to handle I/O
  (non-blocking operations)
- no resource sharing
- primitive interrupt/exception handlers

➡ the default (standalone) config. for Microblaze systems in XPS

Pseudo MPST

- use several processors with SPST
- sharing and synchronization is possible but requires some work
  - PLB bus is common, thus all the peripherals are shared
  - FSL can be used to exchange data
  - mutexes, other hardware IPs are also available
- primitive interrupt/exception handlers

More than a single thread

- Cases b,c,d require support for:
  - ✓scheduling: priorities, queues, timers
  - ✓contexts: TCBs, create, save/restore
  - ✓data sharing/synchronization
    - o locks, shared memory, messages, buffers,…
  - ✓protection: memory spaces, access rights, reentrant code
  - ✓interrupts/exceptions: arithmetic, memory access, I/O, timers, etc.

SPMT

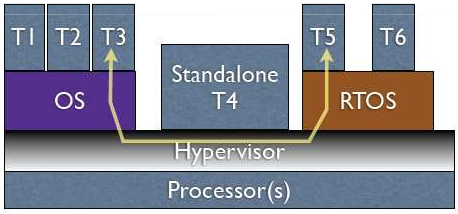
- common in many embedded systems since the applications are fixed, already loaded at boot time!
- file systems, if they exist, are used for data only (images, audio, sensor samples), not programs
- ➡ xilkernel in Xilinx EDK

Many Embedded OS

- Linux/Unix
  - MicroBlaze OSL (Xilinx)
  - PetaLinux, uCLinux (MicroBlaze support)
  - Android ports on the way!?
- RTOS (MicroBlaze support)
  - FreeRTOS
  - Nucleus OS, VxWorks - multicore
  - SynthOS : synthesize your own RTOS (900b footprint)
  - …

HW Virtualization

hypervisor (virtual-machine monitor) allows multiple OS's to run concurrently on the same hardware



HW Virtualization

- advantages:
  - ✓simplified development!
    - o OS developers develop for the abstract hypervisor hardware, not for each specific platform
    - o Hw developers port the hypervisor (microvisor) once, and get the benefit of having many OS running on it
  - ✓more robust systems!
  - ✓drawback: performance penalty

Example: OKL4 microvisor

Handling I/O

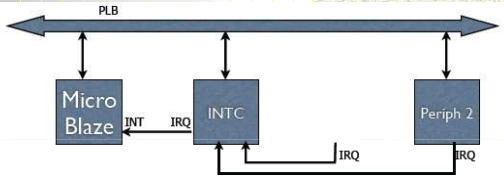
In embedded systems is essential to be able to read and output data!

- polling (usually single threaded apps):
  1. examine the peripheral for new data (non-blocking!)
  2. handle the new data if it exists
  3. move on to other peripherals or activities
  4. go back to 1.
- interrupts (multi-threaded apps):
  1. write a specific function - handler (usually very short)
  2. associate it with the desired peripheral (uses interrupts)
  3. expect "normal code" to be interleaved with handler calls

Accessing I/O

- memory mapped I/O (bus I/O)
  - located within the normal memory address space
  - accessed using regular memory read/writes
  - usually offer both polled or interrupt based handling
  - some may use DMA to transfer data (become bus masters)
  - e.g. xps_gpio, xps_timer on PLB
- special I/O (port I/O)
  - implemented in the processor by specific instructions/lines
  - e.g. FSL on MicroBlaze

MB Interrupts (HW)



A simplified generic interrupt cycle:

- Peripheral asserts IRQ
- CPU detects INT, clears flags
  Handles (also acknowledge IRQ to the peripheral - bus)
- Peripheral de-asserts IRQ

INTC: priority levels, heterogeneous IRQ signals

Device Drivers

- software packages abstracting away certain hardware
  - ✓vary in complexity/functionality
  - ✓low level:
    - o small code library
    - o allow for fine control (register level access)
    - o require extensive development from users
  - ✓high level:
    - o large libraries (and memory footprint)
    - o easy to use API

Device Drivers

- Must comply both Hw and Sw ends!
  - ✓Linux: special way of coding
  - ✓Stadalone-Xilkernel/Microblaze:
    - o no big restrictions since there is no complex OS on top, only your own application
    - o can be included with your custom IP core and distributed along with it for use in EDK
      - – specific directory structure
      - – low level driver generated by the Create IP wizard



Exceptions

- Undesired situations, but from which the system can recover
  - ✓math
    - o divide-by-zero, illegal operand, overflow, underflow,...
  - ✓bus timeouts
  - ✓memory
    - o illegal opcode, unaligned data,...
- These may or may not have handlers within the used framework!

Timing

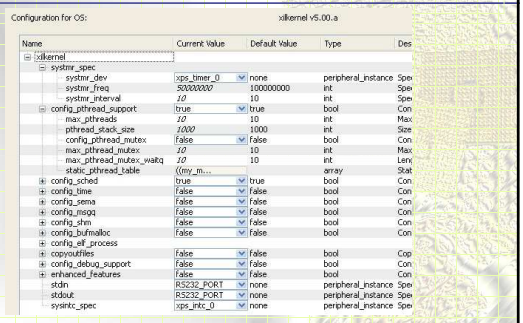
➡ scheduling, performance meters

- ✓lowest level: timers
  - o Hw: usually at least one for the system tick + some WDTs
  - o Sw: many software timers
- ✓higher level: priorities
  - o none (round-robin), static (RM, DM), dynamic (EDF)
- ✓even higher: deadlines, periods
  - o fully rely on the underlaying framework to handle timing

Tools Support

- components (IPs, OS, libs, JVMs)
- component creator (Hw & Sw)
- system builder
- assemblers/compilers (C, C++, Java)
- emulators/simulators
- download/configuration/monitoring
- profilers/debuggers

An Example: Xilkernel



Xilkernel Overview

- POSIX threads API
  - ✓ pthread_create, join, yield, detach, kill,...
  - ✓ round-robin or priority scheduling
- POSIX semaphores
  - ✓ sem_init, destroy, wait, trywait, post,...
- XSI/POSIX message queues
  - ✓ msgget, msgctl, msgsnd, msgrcv
- XSI/POSIX shared memory
  - ✓ shmget, shmctl, shmat, shm_dt
- POSIX mutex locks
  - ✓ pthread_mutex_init, destroy, lock, unlock,...

Xilkernel Overview

- dynamic buffer memory management
  - ✓ faster but less powerful than malloc/free
  - ✓ bufcreate, bufdestroy, bufmalloc, buffree
- software timers
  - ✓ xget_clock_ticks, time, sleep
- exceptions (limited)
  - ✓ registered as faults
  - ✓ faulting threads are killed and the nature of the exception is reported on the console (in verbose mode)
  - ✓ custom handlers cannot be registered
- memory protection (limited)
  - ✓ automatic + user spec., code/data/io violations, TLB

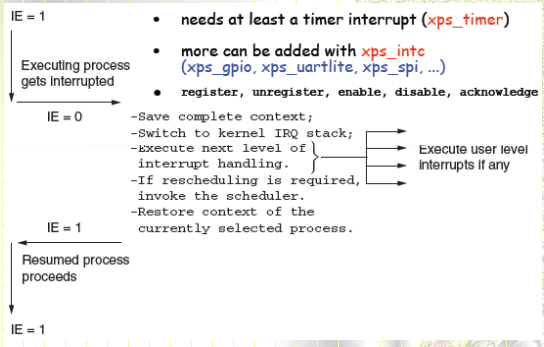
Xilkernel

- **Initialization**
  - kernel entry point `xilkernel_main()` in main.c
  - all user initialization must be done before (set up hardware cores)
- **Thread safety**
  - many library and driver routines are NOT thread safe! (not reentrant, e.g. `printf, sprintf, malloc, free,…`)
  - solution: use locks/semaphores to ensure exclusion
- **Customization**
  - many parameters (max pthreads, semaphores, sched type)
  - many modules may be individually included (saves memory) On MicroBlaze the kernel takes between 7 and 22kb

Using Xilkernel with MB

1. build a system with the XPS wizard
   a. add an `xps_timer`
   b. select to use interrupts for the timer and other peripherals you will handle (this will add `xps_intc` core)
2. create your application
   a. add a new application in the Applications tab, Project view
   b. include "`xmk.h`" and other necessary headers
   c. write your static threads (start on boot) as `void* f(void*)` functions, call `xilkernel_main()` in `main()`
   d. write your needed pthreads program in the static threads
3. configure the software
   a. select `xilkernel` in the Software Platform Settings
   b. configure OS & Lib (STDIN/OUT, timer and intc instance, add modules, clock frequency, table of static threads)
   c. add `-lxilkernel` in Libraries to link against, Set Compiler Options… dialog, Path & Options tab

Xilkernel Interrupts



- needs at least a timer interrupt (xps_timer)
- more can be added with xps_intc (xps_gpio, xps_uartlite, xps_spi, …)
- `register, unregister, enable, disable, acknowledge`

An example with GPIO

```
#include "xmk.h"
#include <stdio.h>
#include <sys/intr.h>
#include "xgpio.h"

// The driver instance for GPIO Device
XGpio ButtonsInput;
// push buttons value
volatile u32 pbValue;

int main(void) {
    // initialize hardware
    InitializeButtons();

    // start xilkernel
    xilkernel_main();
}

void* my_main(void) {
    // xilkernel is started.
    // set up interrupt handlers
    setUpButtonsHandler();

    // enable interrupts in Microblaze
    microblaze_enable_interrupts();

    // a simple loop
    // to check whether the interrupts work
    {
    u32 oldB = pbValue;
    while(1) {
        while(oldB == pbValue) { /* busy wait */ };
        xil_printf("buttons pushed %d", pbValue);
        oldB = pbValue;
    }
    return NULL; // never reached
}
```

```
void InitializeButtons() {
    // initialize GPIO structure
    XGpio_Initialize(&ButtonsInput, XPAR_PUSH_BUTTONS_3BIT_DEVICE_ID);
    // should ALWAYS check the return status!

    //Set the direction for all signals to be inputs
    XGpio_SetDataDirection(&ButtonsInput, 1, 0xFFFFFFFF);

    // enable GPIO interrupts by bit
    XGpio_InterruptEnable(&ButtonsInput, 0xFFFFFFFF);
    // enable GPIO interrupts globally
    XGpio_InterruptGlobalEnable(&ButtonsInput);
}

void setUpButtonsHandler() {
 // associate interrupt channel with handler in INTC
 register_int_handler(XPAR_XPS_INTC_0_PUSH_BUTTONS_3BIT_IP2INTC_IRPT_INTR,
                      buttonsHandler, &pbValue);
 // enable interrupt channel in INTC
 enable_interrupt(XPAR_XPS_INTC_0_PUSH_BUTTONS_3BIT_IP2INTC_IRPT_INTR);
}

void buttonsHandler(void *p) {
    // read data from the GPIO
    *(u32*)p = XGpio_DiscreteRead(&ButtonsInput, 1);
    // clear interrupt in the GPIO
    XGpio_InterruptClear(&ButtonsInput, 0xFFFFFFFF );
    // ack interrupt to the INTC
    acknowledge_interrupt(XPAR_XPS_INTC_0_PUSH_BUTTONS
    _3BIT_IP2INTC_IRPT_INTR);
}
```