Software in Embedded Systems

EDA385: Lecture 4
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!
An 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,...)
A few requirements

Embedded software usually needs to:

1. execute concurrent activities
   (multiprogramming, resource sharing)

2. handle various I/O at different level
   (devices, drivers, polling or interrupts)

3. detect faults, fail gracefully (exceptions)

4. support timing (RT, deadlines, priorities)

5. easy to develop, verify, manage... (tools)
1. Concurrency

- Processes (Tasks)
  - running programs
  - separate code/data
  - few shared resources
  - appear as executing at the same time
  - 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
  - mutexes, other hardware IPs are also available
- primitive interrupt/exception handlers

➡ dual Microblaze in EDAN15 labs
More than one thread

Cases b, c, d require support for:

• scheduling: priorities, queues, timers
• contexts: TCBs, create, save/restore
• data sharing/synchronization
  • 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)
  - ...

2010-09-22
Hw virtualization

- hypervisor (virtual-machine monitor): allows multiple OS’s to run concurrently on the same hardware
Hw virtualization

• advantages:
  • simplified development!
    • OS developers develop for the abstract hypervisor hardware, not for each specific platform
    • 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
2. 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
I/O types

• 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:
    - small code library
    - allow for fine control (register level access)
    - require extensive development from users
  - high level:
    - large libraries (and memory footprint)
    - easy to use API
More Device Drivers

Must comply both Hw and Sw ends!

• **Linux:** special way of coding

• **Standalone-Xilkernel/Microblaze:**
  • no big restrictions since there is no complex OS on top, only your own application
  • 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
3. Exceptions

Undesired situations, but from which the system can recover.

- math
  - divide-by-zero, illegal operand, overflow, underflow,...
- bus timeouts
- memory
  - illegal opcode, unaligned data,...

- These may or may not have handlers within the used framework!
4. Timing

- scheduling, performance meters

• lowest level: timers
  - Hw: usually at least one for the system tick + some WDTs
  - Sw: many software timers

• higher level: priorities
  - none (round-robin), static (RM, DM), dynamic (EDF)

• even higher: deadlines, periods
  - fully rely on the underlaying framework to handle timing
5. Tool 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

OS & Library Settings

OS: xilkernel
Version: 5.00.a

Xilkernel is a simple and lightweight OS.
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_destroy`
- **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
more on 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
Interrupt Handling

Xilkernel abstracts away primary interrupt handling requirements from the user application. Even though the kernel is functional without any interrupts, the system only makes sense when it is driven by at least one timer interrupt for scheduling. The kernel handles the main timer interrupt, using it as the kernel tick to perform scheduling. The timer interrupt is initialized and tied to the vectoring code during system initialization. This kernel pulse provides software timer facilities and time-related routines also. Additionally, Xilkernel can handle multiple interrupts when connected through an interrupt controller, and works with the xps_intc interrupt controller core. The following figure shows a basic interrupt service in Xilkernel.

The interrupt handling scenario is illustrated in this diagram. Upon an interrupt:

- The context of the currently executing process is saved into the context save area.
- Interrupts are disabled from this point in time onwards, until they are enabled at the end of interrupt handling.
- This alleviates the stack burden of the process, as the execution within interrupt, does not use the user application stack.
- This interrupt context can be thought of as a special kernel thread that executes interrupt handlers in order. This thread starts to use its own separate execution stack space.
- The separate kernel execution stack is at least 1 KB in size to enable it to handle deep levels of nesting within interrupt handlers. This kernel stack is also automatically configured to use the pthread stack size chosen by the user, if it is larger than 1 KB. If you foresee a large stack usage within your interrupt handlers, you will need to specify a large value for pthread_stack_size.

This ends the first level of interrupt handling by the kernel. At this point, the kernel transfers control to the second level interrupt handler. This is the main interrupt handler routine of the interrupt controller. From this point, the handler for the interrupt controller invokes the user-specified interrupt handlers for the various interrupting peripherals.

In MicroBlaze processor kernels, if the system timer is connected through the interrupt controller, then the kernel invisibly handles the main timer interrupt (kernel tick), by registering itself as the handler for that interrupt.

Interrupt handlers can perform any kind of required interrupt handling action, including making system calls. However, the handlers must never invoke blocking system calls, or the entire kernel is blocked and the system comes to a suspended state. Use handlers wisely to do minimum processing upon interrupts.

**Figure 5: Basic Interrupt Service in Xilkernel**
#include "xparameters.h"
#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);
}

An example with GPIO
## Xilkernel vs. Standalone

<table>
<thead>
<tr>
<th></th>
<th>standalone</th>
<th>xilkernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>concurrency</td>
<td>--</td>
<td>+++</td>
</tr>
<tr>
<td>cross-development (portability)</td>
<td>-</td>
<td>++</td>
</tr>
<tr>
<td>memory footprint</td>
<td>+++</td>
<td>-</td>
</tr>
</tbody>
</table>