

---

## PRTB Mini Boot Camp Nov 1, 2001 BWRC

Motivation and name courtesy of Culler et. al. & TOS

---

## Topics To Attempt To Cover

- **system** – there are FOUR sides to the story
  - **the Real World** - Node Hardware
  - **the Abstract World** - Four Pieces of the System
- **software model** – programming to the ARM
  - **the workspace** - application entry points
  - **system support libraries for ARM**
- **configurable logic model** – programming to the Xilinx
  - **the workspace** - top level schematic
  - **system support libraries for Xilinx**
- **programming for embedded systems in general and the PRTB**

---

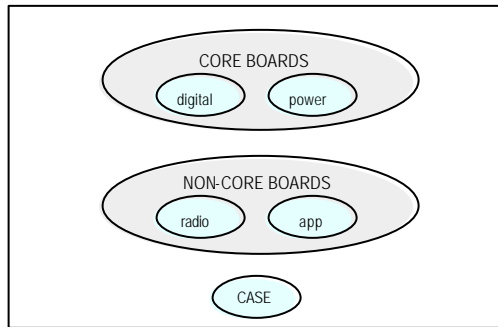
## Topics To Attempt To Cover

- **ARM  $\leftrightarrow$  Xilinx communication**
- **LAB**
  - *writing and compiling a simple program*
  - *loading and running a program*
  - *debugging with or without a laptop*
  - *programming the flash for embedded operations*
- **example network application**

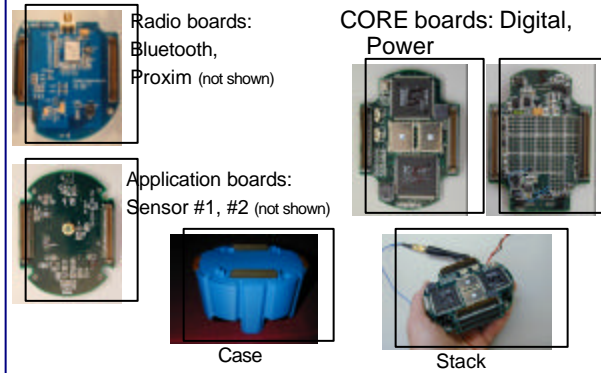
---

## The System

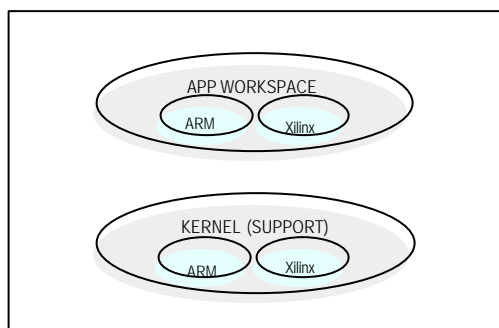
## Node Hardware



## Node Hardware

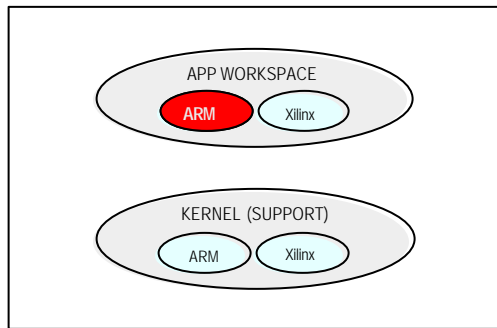


## Building Applications - The Four Pieces



Piece 1:  
ARM Software Model  
- the Workspace

## The Four Pieces – ARM workspace



## the workspace - ARM model

<b>application.c</b>	user entry points app_Init() app_Main() app_RunCommand() app_PrintMenu() app_FIQISR()
<b>application.h</b>	user configuration (Xilinx file location, etc) DEF_XILINX_FILENAME DEF_XILINX_HEADER
<b>main.c</b>	no user serviceable parts inside
<b>parse.c</b>	a command-line parser – hack if you like

## the workspace - ARM model; app\_Init()

```

/*
 * app_Init()
 *
 * Any initialization code goes in here. This function is called once at system
 * start-up. Common usage is for data structure initialization, Xilinx programming, etc.
 */

void app_Init(InitOpts *opts)
{
    /*
     * By default, the Xilinx is automatically loaded with the program pointed to by
     * DEF_XILINX_FILE (Debug variant) or DEF_XILINX_HEADER (Standalone or Rom variants).
     * See application.h in this project. If you do not need a Xilinx design for your
     * application, you can load the default Xilinx program from the WORKSPACE_TEMPLATE
     * directory (DEF_XILINX_FILE and DEF_XILINX_HEADER initially point to this directory).
     */
    /* With the basic Xilinx program, you can control ARM DVS settings using power_SetDVSLevel()
     * and manipulate the eight digital board LEDs using xilinx_WriteLEDs().
     */

    /*
     * REQUIRED if you include TestBed library blocks in an application Xilinx program:
     * App code must inform the operating system of the chosen port assignments for library blocks.
     * For example, if you use a radio library block in your design (e.g. bluetooth_block) and
     * choose port 3 as the radio control port, you must call this function in app_Init():
     * xilinx_SetPortAssignment(PA_RADIO_CNTL, 3);
     * See the ARM API documentation for the requirements and constants for each block.
     */
}
    
```

## the workspace - ARM model; app\_Main()

```

/*
 * app_Main()
 *
 * All user code except for initialization goes in here. app_Main() is part of the
 * operating systems main event loop, so it MUST return. Bottom line: don't use infinite
 * loops in your code. app_Main() is already part of an infinite loop that also does
 * system tasks, including power control. If your code has an infinite loop, the system
 * tasks can't be done.
 *
 * There is one exception to this rule: during development, you might want to do blocking
 * reads with a laptop, for instance in a command interpreter (see systest_RunCommand()
 * below). This is OK for testing but can't be a part of embedded application code.
 */

void app_Main(InitOpts *opts)
{
    // This is a command-line parser for debugging and test. You can remove it if you want.
    systest_RunCommand(opts);
}
    
```

## the workspace - ARM model; menu

```
-----
Commands are case insensitive

run TEST program
  Ts
  Tn[a,p,l,f,i] - Xilinx
                  a - all, p - program, l - LEDs, f - FIFO, i - IRQ

ARM Register read and write
  Rr
  Rw
  Rp
  - print a list of register names

ARM power modes
  Id[le] - Enter ARM IDLE mode (exit via any interrupt)
  Sl[ee] - Enter ARM SLEEP mode (exit via selected interrupt)

Generic Xilinx commands
  U[on] - Read Xilinx UPLINK channel, n = [0-f]
  D[on] - Write Xilinx DOWNLINK channel, n = [0-f]
  Ds
  Ds - Set value of serial data stream block, if instantiated
  Xl[on] - Write to LED channel, [x]1-1 to release LEDs
  Xr[on] - Write to reset channel
  Xz
  Xz - Get board ID (V2+ only)
  Xv
  Xv - Enable external oscillator
  Xp
  Xp - Program, using default raw file (TestShite_PicoRadioBoards.raw)
  Xp <filename> - Program, using specified raw file
  Xr - Read from SRAM (not supported yet)
  Xw - Write to SRAM (not supported yet)
  Xr - Read from FLASH (not supported yet)
  Xr - Reset FLASH (not supported yet)
  Xe - Erase FLASH (not supported yet)
```

## the workspace - ARM model; menu cont.

```
Bluetooth control
  Bt
  Bp<>-> - Initialize controller (power on, start)
  Bt<>-> - Power on, off, release
  Bt<>-> - Transmit on, off, release
  Br<>-> - Receive on, off, release
  Bcr<>-> - Set RX_ON, clear TX_ON, release TX_ON
  Bk<>-> - Hop mode on, off, release
  Bk0n - Load channel n
  Bk1 - Load register
  Bk - Get raw RSSI sample
  Bk - Get current channel
  Bk - Read status
  Bv - Set console verbose level

Proxim control
  Pp<>-> - Power on, off, release
  Pp<>-> - Standby on, off, release
  Pt<>-> - Transmit on, off, release
  Pr<>-> - RSSI sampling on, off, release
  Pk - Get RSSI sample
  Pk[n] - Set Proxim to receive channel n, and receive
  Pk[n] - Set Proxim to transmit channel n, and transmit
  Pk - Release channel control
  Pk<>-> - Console verbose mode on, off, off

Generic commands
  Cc<>-> - Print channel library trace info for node n
  S<>-> - Set supply power on, off
  V - Set Verbose level
  Wd - Print menu to project directory
  Q - Exit
  Z - Toggle menu

main:
```

## the workspace - ARM model; commands

```
/*
 * app_RunCommand()
 *
 * This function is executed at the start of systest_RunCommand(). Its purpose is to give
 * you the ability to add custom commands. The switch looks at the first character in the
 * string entered at the console prompt generated by systest_RunCommand() - any other characters
 * can be interpreted locally. To add a command, add a case to the switch with the command
 * letter in single quotes, upper case, for example 'M'. To determine which letters are
 * already used by systest_RunCommand(), run the ARM program that comes with the workspace
 * template in the ARMulator (an option of the ARM debugger). Type in the letter Z at the
 * command prompt. This will print a menu to the screen.
 *
 * Return values: The return value from app_RunCommand() determines what systest_RunCommand()
 * does after app_Command() returns. If the return value is 0, systest_RunCommand() will
 * return without examining the command string. If the return value is -1, systest_RunCommand()
 * will do its usual command interpretation. For your custom commands, you will typically
 * choose a command letter that is not already interpreted by systest_RunCommand(). However,
 * you can create a subcommand (second letter in the command string) of an existing command
 * by looking for the first level command letter in the app_RunCommand() switch, check the
 * second letter in the command string, and return -1 if the second letter doesn't match any
 * of your custom subcommands. This will cause systest_RunCommand to continue normal command
 * detection based on the first letter in the command string.
 */

int app_RunCommand (InitOpts *opts, char *arglist)
{
  switch (toupper(arglist[0])) {
    default:
      return -1;
  }
  return 0;
}
```

## the workspace - ARM model; custom menu

```
/*
 * app_PrintMenu()
 *
 * This is a companion function to app_RunCommand(). If you want to add a line to the
 * systest_RunCommand() menu, put it here. This function is called by systest_RunCommand()
 * at entry, so custom commands will be displayed at the top of the default menu.
 */

void app_PrintMenu(FILE *fh)
{
  fprintf(fh, "\n");
}
```

## the workspace - ARM model; app\_FIQISR()

```
/*
 * app_FIQISR()
 * Put FIQ interrupt service routing code in this function. Code must be in-line
 * (no function calls). Use macros instead. If you have to call functions, use
 * an IRQ.
 *
 * IRQ interrupt service routines (ISRs, or handlers) are per device.
 * See the interrupt documentation. There is ONLY ONE User-level FIQ ISR.
 */
__irq void app_FIQISR(void)
{
    inter_level = 1;
    // PUT CODE HERE
    inter_level = 0;
}
```

## the workspace - ARM model; application.h

```
/*.....*/
//
// application.h
//.....*/

#ifndef _APPLICATION_H_
#define _APPLICATION_H_

#include <stdlib.h>
#include <stdio.h>

#include "kernel.h"
#include "datastructs.h"
#include "xillo.h"

/* Xilinx port assignments */

#define DEF_XILINX_FILENAME "H:/PicoRadio/TestBed/Support/WORKSPACE_TEMPLATE/Xilinx/compiled/toplevel_template.raw"
#ifdef STANDALONE
#define DEF_XILINX_HEADER "H:/PicoRadio/TestBed/Support/WORKSPACE_TEMPLATE/Xilinx/compiled/toplevel_template.h"
#endif

void app_Init(InitOpts*);
void app_Main(InitOpts*);

#endif
```

## the workspace - ARM model; main.c

```
extern InitOpts opts;
InitOpts opts;

main(int argc, char **argv)
{
    _init_opts(&opts);

    parse_Commands(&opts, argc, argv);

    kernel_InitSys(&opts);

    parse_PrintOptions(&opts);

    app_Init(&opts);

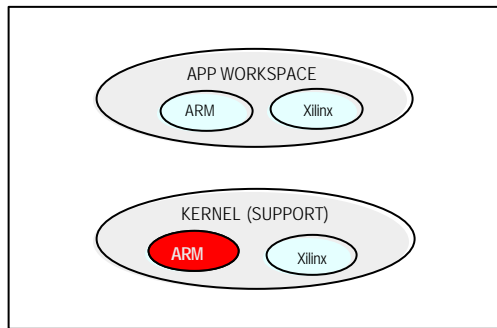
    while (1) {
        app_Main(&opts);
        kernel_IdleLoop(&opts);
    }

    return 0; // not reached
}
```

---

Piece 2:  
ARM Software Model  
- system support libraries

## The Four Pieces – ARM kernel support



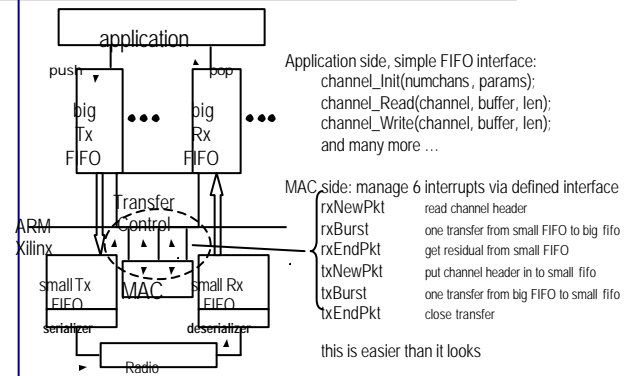
## kernel support – ARM libraries

- armos
- channels
- flash
- FPGA
- radios

## kernel support – libraries; armos

kernel	system initialization, idle control
interrupt support	install handlers/enable/disable
OS timers	4 ARM timers in ms resolution
power control	DVS, radio on/off, Xilinx idle
serial1	FIFO interface to RS232
Serial4	special-purpose serial ports (SSP, codec)
ARM GPIO support	~12 general-purpose ARM I/O pins
data structures	double-buffered FIFO, dlls, p-queue
systest	systest_RunCommand()
misc algorithms	u-law, ...
debug	debug printf management

## kernel support – libraries; channels



## kernel support – libraries; ARM flash

flash.exe                    menu driven Windows program with all of the below

int flash\_Read();            read n memory locations

int flash\_Write();           write one memory location

int flash\_WriteBuf();        write n memory locations

int flash\_Erase();           erase entire flash

int flash\_EraseSector();    erase one sector

int flash\_Program();        erase entire flash, then program from file

## kernel support – libraries; FPGA

low-level Xilinx I/O        xilinx\_ReadPort(port);  
                              xilinx\_WritePort(port, data);  
                              xilinx\_SetPortAssignment(port, service);  
                              xilinx\_WriteLEDs(value);  
                              xilinx\_GetID();  
                              xilinx\_Reset(signal);

datapath block support     datapath\_LineBalance(onoroff);  
                              datapath\_CRC(onoroff);  
                              datapath\_ReadControlByte();  
                              datapath\_WriteControlByte();  
                              datapath\_ReadDataByte();  
                              datapath\_WriteDataByte();

TDMA block support        tdma\_Init(numslots, slotlen, prelen, ...);  
                              tdma\_Config(numslots, slotlen, prelen);  
                              tdma\_AddRxSlot(slot);  
                              tdma\_AddTxSlot(slot);  
                              tdma\_RemSlot(slot);  
                              tdma\_SetBasestation();  
                              tdma\_SetRemote();



## kernel support – libraries; radios

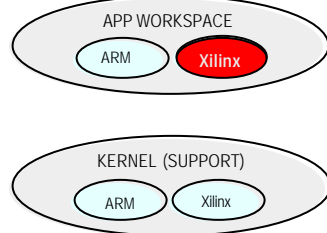
bluetooth                  bluetooth\_Init();  
                              bluetooth\_Rx(onoroff, ...);  
                              bluetooth\_Tx(onoroff, ...);  
                              bluetooth\_Power(onoroff, ...);  
                              bluetooth\_SetChannel(channel#);  
                              bluetooth\_GetRSSISample();

proxim                      proxim\_Init();  
                              proxim\_Tx(onoroff, ...);  
                              proxim\_Power(onoroff, ...);  
                              proxim\_Standby(onoroff, ...);  
                              proxim\_SetChannel(channel#);  
                              proxim\_GetRSSISample();



Piece 3:  
Configurable Logic Model  
- the Workspace

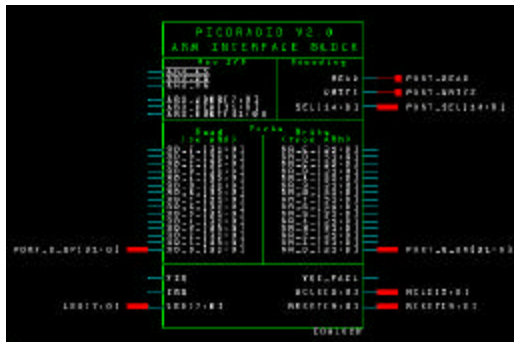
## The Four Pieces – Xilinx workspace



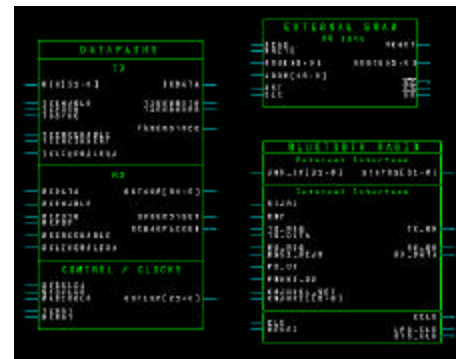
## the workspace - Xilinx model; toplevel\_template



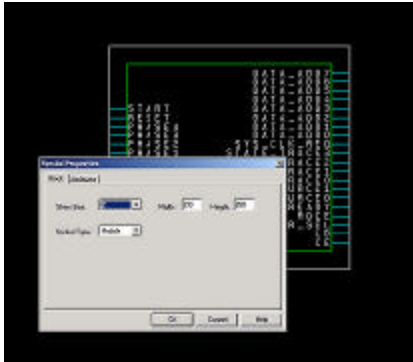
## the workspace - Xilinx model; ARM Interface



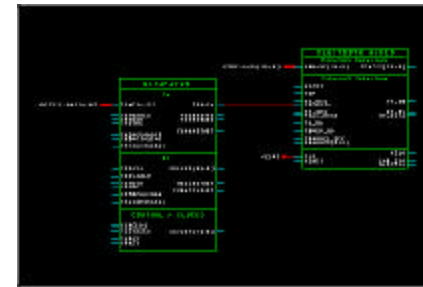
## the workspace - Xilinx model; library blocks



the workspace - Xilinx model;  
embedded VHDL

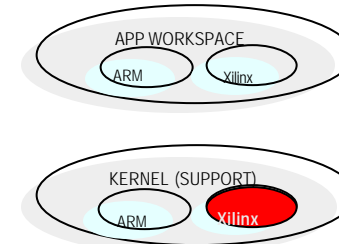


the workspace - Xilinx model;  
connections



Piece 4:  
Configurable Logic Model  
- system support libraries

The Four Pieces – Xilinx kernel support



## kernel support – Xilinx libraries

- palette
- bluetooth
- datapaths
- FIFO
- I/O
- misc
- proxim
- state
- TDMA
- xc4000x
- builtin

## kernel support – Xilinx libraries; palette



## Programming for Embedded Systems in General and Issues with the PRTB

## programming for embedded systems

- Compared to non-embedded systems, embedded systems:
  - Have no console
  - Have limited hardware resources
  - Have limited or no operating system support
  - Have limited or no debug support
  - Run only simple applications
  - Are generally event-driven
  - **Rely heavily on asynchronous events (watch out for resource conflicts!)**
- So, in an embedded system, you generally can't:
  - Wait for user input from a keyboard, mouse, or other standard input device.
  - Use standard C-lib function calls like printf and scanf
  - Debug with break points or by stepping a program
  - Access stored data directly

## programming for embedded systems

- What is an event?
  - Timer goes off when it's time to get a new sensor sample
  - Wake-up timer goes off when the CPU is sleeping
  - Protocol needs service e.g. a new packet has arrived
  - Buffer is empty, almost empty, full, almost full, half full, ...
  - Byte ready to transfer on serial port
  - And many more..
- These are the PRTB interrupt sources:
  - Two Xilinx interrupts                   MAC, mainly
  - GPIO pin interrupts                   sensor board or software self-interrupt
  - Timer interrupts                       events needing timing with ms precision
  - Real-time clock interrupts           events synchronized with the system clock
  - Serial port interrupts                sensor board, PC during debugging

## embedded programming issues for PRTB, p1

- The PRTB is a single-threaded event-driven embedded system
- There are TWO ways to detect events:

**Polling:** in a loop, query status from some event source of sources

Pros:

Simple implementation.  
Low overhead.  
Synchronous. Can manage shared resources more easily and safely.

Cons:

CPU must run on a regular basis, regardless of whether there is an event to service or not. Can't take full advantage of power control techniques.  
Average delay in response to an event is roughly half the poll interval.

**Interrupts:** when an event occurs, immediately cause the CPU to handle the event

Pros:

Low event service latency.  
Can idle or put the CPU to sleep between events. The interrupt will wake the CPU.  
Less code in main loop.

Cons:

Can be complex to implement.  
Asynchronous. Must protect shared resources from contention (multiple reader single writer). Never know when your main code will be interrupted.

## embedded programming issues for PRTB, p2

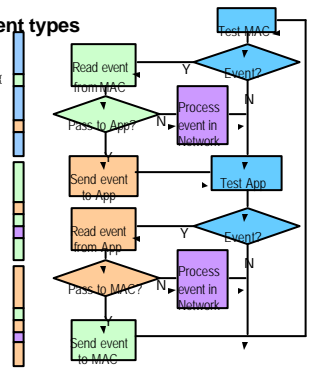
### Example of Polling with two event types

```

void app_Main(InitOpts *opts)
{
    if ((event = xilinx_ReadPort(MACSTATUS) & MACEVENTMASK) {
        mac_GetEvent(event);
    }
    if ((event = app_GetStatus(APPSTATUS) & APPEVENTMASK) {
        app_GetEvent();
    }
}

void mac_GetEvent(irevent)
{
    if ((data = xilinx_ReadPort(MACPORT) & PASS_TO_APP) {
        app_TakeEvent(event);
    } else {
        network_ProcessEvent(data);
    }
}

void app_GetEvent(irevent)
{
    if ((data = app_GetData()) & PASS_TO_MAC) {
        mac_TakeEvent(event);
    } else {
        network_ProcessEvent(data);
    }
}
    
```



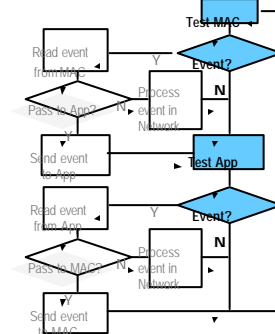
## embedded programming issues for PRTB, p3

### Example of Polling

what's wrong with this picture?

Even if both event tests = 'N', main thread must spin

- CPU can't idle



## embedded programming issues for PRTB, p4

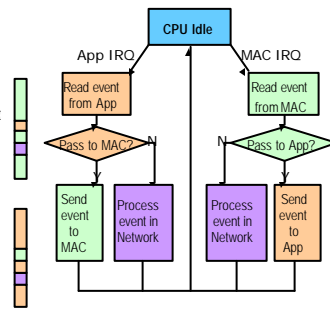
### Example of Interrupts – two event types

```

void app_Main(InitOpts *opts)
{
}

void mac_GetEventInterruptHandler(int event)
{
    if ((data = xilinx_ReadPort(MACPORT) & PASS_TO_APP) {
        app_TakeEvent(event);
    } else {
        network_ProcessEvent(data);
    }
}

void app_GetEventInterruptHandler(int event)
{
    if ((data = app_GetData()) & PASS_TO_MAC) {
        mac_TakeEvent(event);
    } else {
        network_ProcessEvent(data);
    }
}
    
```



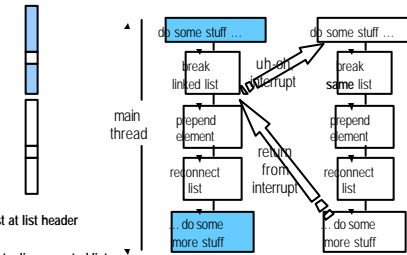
## embedded programming issues for PRTB, p5

### Example of Interrupts – how to screw yourself up really easily

```

void app_Main(InitOpts *opts)
{
    do some stuff ...
    dl_Prepnd(list, data);
    ... do some more stuff
}

void SomeInterruptHandler()
{
    do some stuff ...
    dl_Prepnd(samelist, newdata);
    ... do some more stuff
}
    
```



1. Main thread disconnects list at list header
2. Interrupt occurs
3. Interrupt handler disconnects disconnected list
4. Interrupt handler reconnects list with new element
5. Main thread reconnects connected list with new element
6. Boom

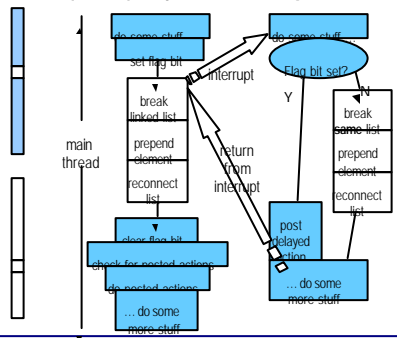
## embedded programming issues for PRTB, p6

### Example of Interrupts – one (crude) way to avoid the problem

```

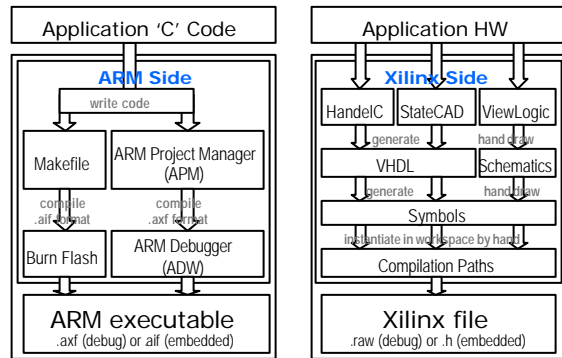
void app_Main(InitOpts *opts)
{
    do some stuff ...
    flagbit = 1;
    dl_Prepnd(list, data);
    flagbit = 0;
    if (posted actions) {
        do posted actions
    }
    ... do some more stuff
}

void SomeInterruptHandler()
{
    do some stuff ...
    if (flagbit == 1) {
        post-delayed action
    } else {
        dl_Prepnd(samelist, newdata);
    }
    ... do some more stuff
}
    
```

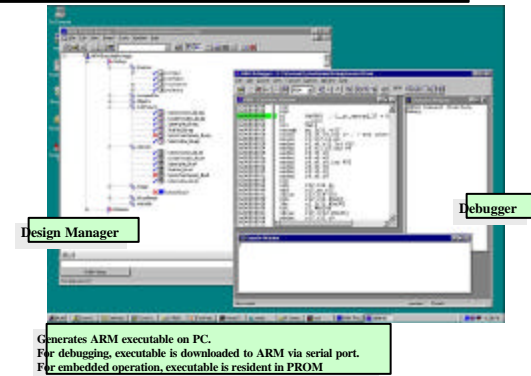


## LAB: Building a Simple Program

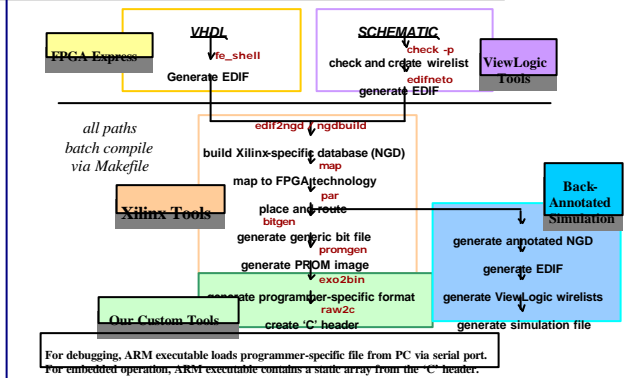
## design flow



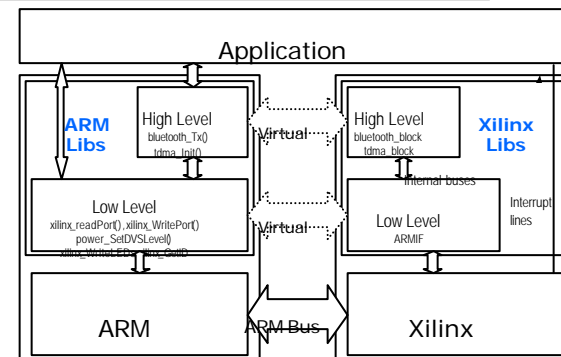
## ARM build environment



## Xilinx compilation – makefile driven

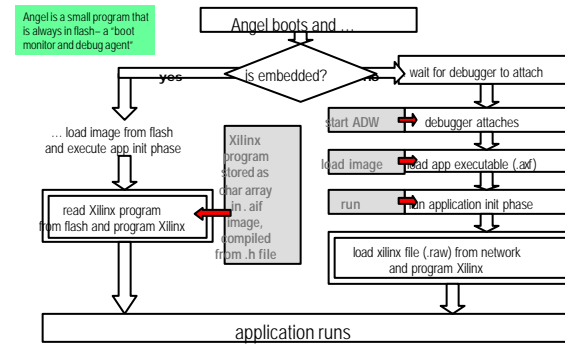


## ARM ↔ Xilinx communication



## LAB: Loading and Running a Program

## run time

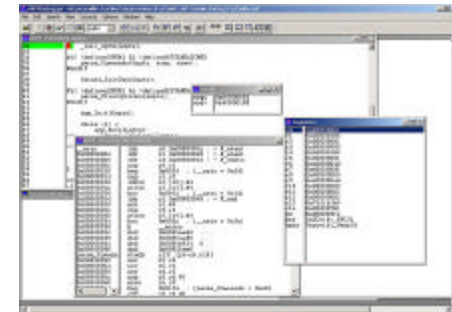


## LAB: Debugging With or Without a Laptop

## debugging with a laptop

- When connected to a computer you can

- Step through a program
- Set breakpoints
- View registers and memory
- View local variables




## debugging without a laptop

- When not connected to a computer you can
  - See the LEDs (yippee)
  - Capture ARM memory bus activity on a logic analyzer using debug tags
  - Extract trace data from a wireless node

Example of debug tags

```
app_TimerISR (int is_really_an_int_addr)
{
    int *addr = (int *) is_really_an_int_a
    volatile char *tagaddr;
    tagaddr = (unsigned *) 0x08000024;
    *tagaddr = 0xf51;
    ostimer_Clear(1);
    *addr += 1;
    ostimer_Resume(1, 500);
}
```



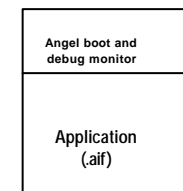
## LAB: Programming the Flash for Embedded Operation

## programming the ARM flash memory

- To be embedded, an image must eventually end up in flash memory.
- Flash is non-volatile i.e. image is not lost between power cycles.
- An embedded image boots and runs on power-up w/o user intervention.

## programming the ARM flash memory

- A PRTB image looks roughly like this:
- In PRTB, Angel copies itself and the app out of flash and into RAM on boot, so the flash can be written w/o effecting the any running code.
- In fact, once the node is booted, you can remove the flash completely and the app will not be affected.



## programming the ARM flash memory

- Programming the flash is done in 4 steps:
  1. Boot the node – image is copied into RAM by Angel
  2. Load a special-purpose app from a laptop using the ARM debugger.
  3. Run the new app – it reads a new image from file and reprograms the flash with the new image.
  4. Reboot the node – the new app will run.

## programming the ARM flash memory

- Program using flash.bat. You get a menu like this:

```
.....  
Commands are case INsensitive  
E - erase flash and load a new image from file  
R - read one word  
Rn - read multiple words  
W - write one word  
Wn - write multiple words  
Ea - erase one sector by address  
Es - erase one sector by sector number  
Ea - erase all sectors  
Z - toggle menu  
Q - quit  
Enter choice |
```

- flash.bat comes with the ARM workspace in ARM/main. Run it from a tcsh shell.

## Example Network Application, ARM Side

## application.c: app\_Init()

```
void app_Init(InitOpts *opts)  
{  
    /*  
     * Basic Xilinx programming and initialization.  
     */  
  
    xilinx_SetPortAssignment(PA_RADIO_CNTRL, 0);  
    xilinx_SetPortAssignment(PA_RADIO_STATUS, 0);  
    xilinx_SetPortAssignment(PA_MAC_CNTRL, 1);  
    xilinx_SetPortAssignment(PA_MAC_STATUS, 1);  
    xilinx_SetPortAssignment(PA_MAC_CONFIG, 2);  
    xilinx_SetPortAssignment(PA_MAC_RXTBL, 3);  
    xilinx_SetPortAssignment(PA_MAC_TXTBL, 4);  
    xilinx_SetPortAssignment(PA_DATAPATH_STATUS, 2);  
    xilinx_SetPortAssignment(PA_DATAPATH_DATA_DN, 5);  
    xilinx_SetPortAssignment(PA_DATAPATH_DATA_UP, 5);  
    xilinx_SetPortAssignment(PA_CHANNEL_EVENT, 10);  
  
    printf("Xilinx init...\n");  
    if (xilinx_Program(opts) < 0) {  
        exit(-1);  
    }  
    xilinx_EnableClock();  
    xilinx_InitializeFIQ(channel_FIQISR);  
    xilinx_InitializeIRQ(app_IRQISR, 0);  
  
    // Initialize Xilinx  
    // Load Xilinx program.  
    // Start Xilinx 200MHz oscillator  
    // Install a Xilinx FIQ handler  
    // Install a Xilinx IRQ handler
```

## application.c: app\_Init()

```
/*
 * Proxim radio controller initialization (Xilinx block and ARM).
 */

printf("Proxim_init...\n");           // Initialize Proxim radio
bluetooth_SetVerbose(ON);
bluetooth_Power(ON);                 // Turn on BT supply
bluetooth_Standby(OFF);
bluetooth_SetChannel(6, -1);         // Set default channel

/*
 * Enable ARM interrupt control module for Xilinx interrupts.
 */

printf("Enabling interrupts...\n");   // Enable interrupts
xilnx_EnableFIQ();                  // Unmask FIQ in ARM controller
xilnx_EnableIRQ();                  // Unmask IRQ in ARM controller
```

## application.c: app\_Init()

```
/* Initialize TDMA controller (Xilinx block and ARM driver) */
printf("Setting up TDMA...\n");       // Initialize TDMA
tdma_Init(NUM_SLOTS, SLOT_LEN, PRE_LEN, 0, 0); // Set channel parameters

#ifdef BASESTATION
printf("\nConfigured as Target\n");
tdma_SetBasestation();
for (int i; for (i = 2; i < NUM_SLOTS; ++i) { tdma_AddRxSlot(i);}}

basestation_init(opts);
#else /* ANCHOR */
printf("\nConfigured as Anchor #d\n", opts->unit_id);
tdma_SetRemote();
tdma_AddTxSlot(opts->unit_id);       // Set Tx slot to be remote ID.

sensor_init(opts);
#endif
} /* end app_Init() */
```

## application.c: app\_Main()

```
void app_Main(InitOpts *opts)
{
#ifdef BASESTATION

    basestation_main(opts);
#endif

    // NOTE: no sensor main - sensor app is completely interrupt driven
} /* end app_Main() */
```

## basestation.c: basestation\_init()

```
void basestation_init(InitOpts *opts)
{
    memset((char *) numsamples, 0, NUM_SLOTS * sizeof(int));
    memset((char *) sd, 0, sizeof(sd));

#ifdef FLASHLOG
    int i;
    for (i = 4; i <= FLASH_LASTSECTOR; ++i) {
        flash_EraseSector(i, 1);
    }
#endif
}
```

## basestation.c: basestation\_main() p1

```
void basestation_main(InitOpts *opts)
{
    int i;

    if (numcycles >= SAMPLES_PER_NODE) {
        memset((char *) dataset, 0, SAMPLE_SET_SIZE_BYTES);

        for (i = 2; i <= NUM_SLOTS; ++i) {
            if (realindx[i] >= 0) {
                int indx = realindx[i] * SAMPLE_SIZE;
                dataset[indx+0] |= ((i & 0x1f) << 27) |
                    ((sd[i].goodsamples & 0xff) << 1) |
                    (sd[i].tmp & 0x3ffff);
                dataset[indx+1] = sd[i].hum;
                dataset[indx+2] = sd[i].lgt;
                dataset[indx+3] = (timestamp + 1) / FRAMES_PER_SEC;
            }
        }

        memset((char *) sd, 0, sizeof(sd));
    }
}
```

## basestation.c: basestation\_main() p2

```
#ifdef FLASHLOG // LOG TO FLASH

static unsigned addr = FLASH_ADDR_APP;
static unsigned limit = FLASH_ADDR_LIMIT - SAMPLE_SET_SIZE_BYTES;

if (addr < limit) {
    flash_WriteBuf(addr, (unsigned *) dataset, 125, 0);
    addr += 4;
}

memset((char *) dataset, 0, SAMPLE_SET_SIZE_BYTES);
```

## basestation.c: basestation\_main() p3

```
#else // OR, LOG TO FILE FOR MATLAB GRAPHING

memcpy((char *) &dumpbuf[bufpos], (char *) dataset, SAMPLE_SET_SIZE_BYTES);
bufpos += SAMPLE_SET_SIZE;

if (bufpos >= DUMP_SIZE) { // if dump threshold
    logfn = fopen(log_name, "ab"); // open log
    if (!logfn) {
        printf("***Error in opening '%s'\n", log_name);
        return;
    }
    fwrite(dumpbuf, DUMP_SIZE_BYTES, 1, logfn); // log to file
    fclose(logfn); // close log
    bufpos = 0;
}

numcycles = 0;
#endif
return;
} /* end basestation_main() */
```

## basestation.c: Interrupt Service Routine

```
void app_IRQISR(unsigned data)
{
    status = channel_GetStatus(); // determine reason
    channel = status & 0x1f; // determine channel

    if (status & CHANNEL_TX_ENDPKT) { // if just transmitted
        channel_ClearStatus(CHANNEL_TX_ENDPKT);
        ++timestamp;
        ++numcycles;
        xilinx_WriteLEDS(timestamp);
        channel_Write(0, (byte *) &slotlen, sizeof(int));
    }
    else if (status & CHANNEL_RX_ENDPKT) { // if just received
        channel_ClearStatus(CHANNEL_RX_ENDPKT);

        channel_Read(channel, (byte *) onesample, sizeof(onesample));

        tmp = onesample[0]; hum = onesample[1]; lgt = onesample[2];
        ++numsamples[channel];
    }
    if (isValid(tmp, hum, lgt)) {
        sd[channel].tmp += tmp;
        sd[channel].hum += hum;
        sd[channel].lgt += lgt;
        ++sd[channel].goodsamples;
    }
} /* end app_IRQISR for basestation */
```

## sensor.c: sensor\_init()

```
void sensor_init(InitOpts *opts)
{
    xilinx_WritePort(11, 0xe3f);           // set XIO pin values for sensor board
    ssp_InitMicrowire(0xb, 0, 0, NULL, 0); // set SSP interface for Microwire
} /* end of sensor_init() */
```

## sensor.c: data acquisition

```
static int get_temperature(void)
{
    float voltage = binary_to_voltage(get_binary_data(0x10));
    return (int) (((voltage - 0.378) / 0.04242) * 100);
}

static int get_humidity(void)
{
    float voltage = binary_to_voltage(get_binary_data(0x50));
    return (int) (((voltage/((float) ADC_REF_VOLTAGE))
        - 0.16) / 0.0062) * 100);
}

static int get_light(void)
{
    const float r_fb = 0.20*10e+3;           // feedback resistor value
    const float int_const = 1.1*10e-9;      // amps per lux of light

    float voltage = binary_to_voltage(get_binary_data(0x20));
    return (int) (((voltage/(r_fb*int_const))) * 100);
}

static unsigned get_binary_data(unsigned short channel)
{
    ssp_push((channel << 8) | 0x8f00);      // push control byte into xmit FIFO
    while (!(xilinx_ReadPort(0) & 0x1));   // if CS* low, do not pop rev FIFO
    return ssp_pop();                       // read value from recv FIFO
}
```