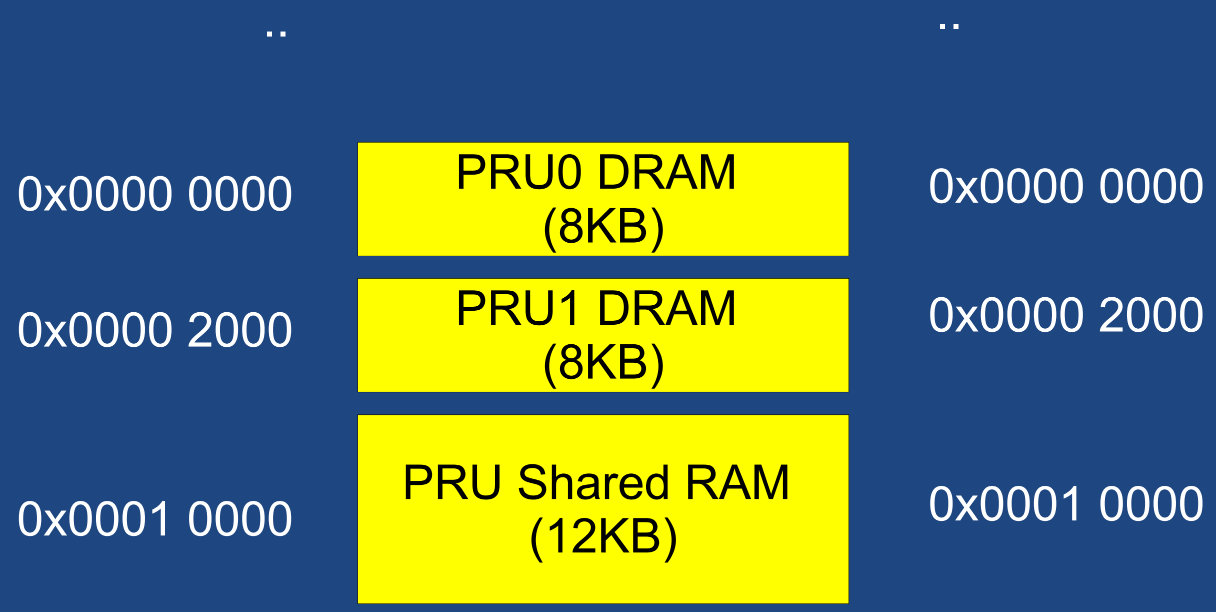
Transferring data between PRU <==> Linux

Topics

1) How we share data between Linux and the PRU

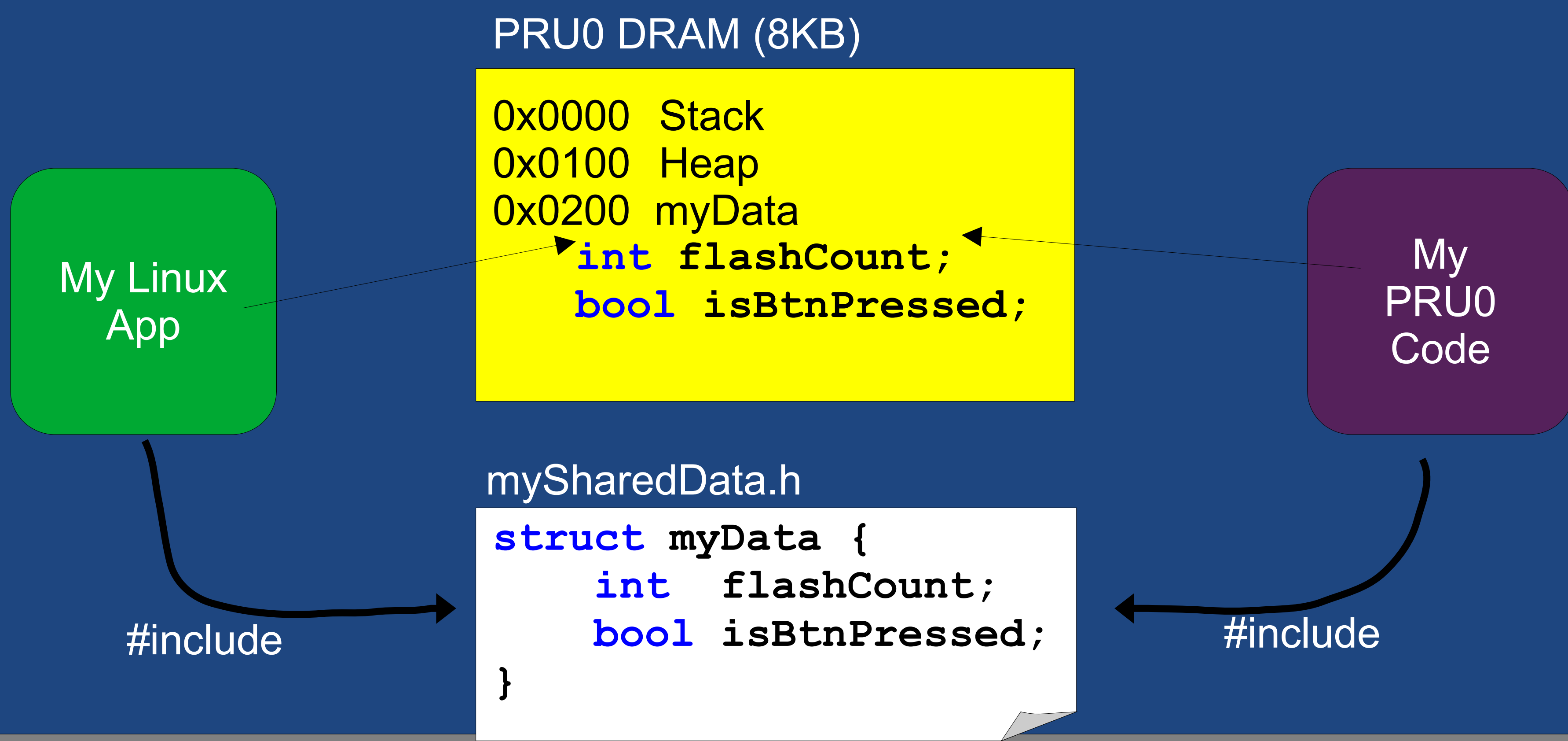
Memory sharing



- Linux global address 0x4a30 0000 base
  - Must be mapped into your app's memory space with mmap()
- PRU1 has same map as PRU0, except:
  - 0x0000 0000 for PRU1 DRAM
  - 0x0000 2000 for PRU0 DRAM

Memory Use

- Shared Memory Idea
  - Directly put values into PRU's memory to share values
  - Hint:..



Sample Program - Shared Struct

- Shared .h file
  - Create one .h file which defines
    ..
    between PRU & Linux
  - Each program #include this same file

```
typedef struct {
  bool isLedOn;
  bool isButtonPressed;
} sharedMemStruct_t;
```

sharedDataStruct.h

Sample Program - PRU

```
#define THIS_PRU_DRAM 0x00000
#define OFFSET        0x200

volatile sharedMemStruct_t *pSharedMemStruct =
  (volatile void *)(THIS_PRU_DRAM + OFFSET);

void main(void){
  // Initialize at startup
  pSharedMemStruct->isLedOn = true;
  pSharedMemStruct->isButtonPressed = false;

  while (true) {
    // Drive LED from shared memory
    if (pSharedMemStruct->isLedOn) {
      __R30 |= LED_MASK;
    } else {
      __R30 &= ~LED_MASK;
    }

    // Sample button state to shared memory
    pSharedMemStruct->isButtonPressed =
        (__R31 & BUTTON_MASK) != 0;
  }
}
```

```
typedef struct {
  bool isLedOn;
  bool isButtonPressed;
} sharedMemStruct_t;
```

sharedDataStruct.h

Sample Program - Linux (1/2)

```
#define PRU_ADDR   0x4A300000  // Start of PRU mem
#define PRU0_DRAM     0x00000 // Offset PRU0 mem
#define PRU_SHAREDMEM 0x10000 // Offset shared mem

// Return the address of the PRU's base memory
volatile void* getPruMmapAddr(void)
{
  int fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (fd == -1) {...}

  // Points to start of PRU memory.
  volatile void* pPruBase = mmap(
      0, PRU_LEN,
      PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, PRU_ADDR);

  if (pPruBase == MAP_FAILED) {
      ...
  }
  close(fd);

  return pPruBase;
}
```

- getPruMmapAddr()
  - Calls mmap() to map the physical PRU memory into our virtual address space.
- freePruMmapAddr()
  - Cleans up when done

```
void freePruMmapAddr(
        volatile void* pPruBase)
{
  if (munmap((void*) pPruBase, PRU_LEN)) {
     perror("PRU munmap failed");
     exit(EXIT_FAILURE);
  }
}
```

Sample Program - Linux (2/2)

```
#define PRU0_MEM_FROM_BASE(base) \
    ( (base) + PRU0_DRAM + 0x200)

volatile void* getPruMmapAddr(void) {..}

int main(void) {
   // Get address to PRU0 memory
   volatile void *pPruBase = getPruMmapAddr();
   volatile sharedMemStruct_t *pSharedPru0
       = PRU0_MEM_FROM_BASE(pPruBase);

   for (int i = 0; i < 20; i++) {
      // Drive LED
      pSharedPru0->isLedOn = (i % 2 == 0);

      // Print button
      printf("Button: %d\n",
         pSharedPru0->isButtonPressed);

      sleep(1);
   }

   // Cleanup
   freePruMmapAddr(pPruBase);
}
```

```
typedef struct {
  bool isLedOn;
  bool isButtonPressed;
} sharedMemStruct_t;
```

sharedDataStruct.h

- Linux app uses pSharedParu0 as though it points to its own struct..
- Must run as root to call mmap()

Demo: Exchange data about Flash & Btn

- See sharedMem:
  - sharedMem-Linux/, sharedMem-PRU/
- Structure
  - Folder for PRU, and for Linux app
  - Shared .h file somewhere
  - Prj root folder makefile copies PRU code, build Linux
- Build/Run Process
  - make on host project root folder to compile and copy code
  - make on target to build PRU code
  - make installPRU0 to run PRU
  - /mnt/remote/myApps/sharedMem-linux to run on target

Packing Structs

Data Types

- C data types can be of different sizes
  - C spec simply mentions their relative size
  - PRU and Linux use:
    1 byte: char
    2 bytes: short
    4 bytes: int, long, float
    8 bytes: long long, double
- ..
  - Gives integer data types based on #bits
  - Useful for..
    uint8_t, uint16_t, uint32_t, uint64_t
    int8_t, int16_t, int32_t, int64_t

Structs

- Structs store different types of data in one allocated unit of memory

```
struct bigBadWolfData_t {
   char numPuffs;
   bool hasBigTeeth;
   int numCookiesEaten;
};
```

- How does this layout in memory?

..

numPuffs
hasBigTeeth
numCookiesEaten

..

numPuffs
hasBigTeeth
numCookiesEaten

2 Processors

- Cortex A8 (Linux) aligns values

  Incorrect alignment gives a bus error
- Word align int/uint32_t
- Double world align doubles, long long, uint64_t
- ..

Padding Structs

```
struct bigBadWolfData_t {
   char numPuffs;
   bool hasBigTeeth;
   char _pad1, _pad2;
   int numCookiesEaten;
};
```

Padded

numPuffs
hasBigTeeth
_pad1
_pad2
numCookiesEaten

Padding bytes

- Add extra bytes to struct..

  char/bool: byte aligned

  int/uint32_t: word aligned

  double/uint64_t: dword aligned

- Once padded correctly, struct is identical as both packed (on PRU) and unpacked (on Cortex A8)
  - Incorrect padding means values written to a field by one processor not seen correctly by other.

Exercise

- Modify sharedMem example
  - Store # PRU loops:
    Add new uint64_t field to struct
    - PRU initialize field, and increment after each loop
    - Make Linux print it
- Experiment:
  - What happens when struct is unpadded?
  - What happens when struct is padded?
  - What happens with 2-byte fields? Need to be aligned on 4 byte? 2 byte?

Troubleshooting

- Hard to debug the PRU because ..
  - Write very little code at a time, then test it.
  - Flash the LED for some visual status
- Common Issues
  - Permission denied on /dev/mem:
    run with sudo
  - Input/output not working:
    check you have run config-pin
  - Data exchange problems:
    check your data structure is word/dword aligned
  - Changes to code not running:
    add compile-time error to check if correct code is compiling

Exercise: Sample GPIO

- Implement the following using a PRU
  - PRU samples GPIO to memory
    (one frame ~2s?)
  - PRU show start of sampling to a frame by toggling LED
  - Linux prints each frame to the screen
    Ex: _XX____XXX_X_XX___XXX___XX
  - Mark frame with a bit to indicate:
    0: it’s free for PRU to fill
    1: it's been filled and ready to be processed by Linux
- Advanced: Double-buffer the frame
  - Have 2 frames
  - PRU fills frame 1 and sets flag, then does frame 2...
  - Linux sees frame is read, processes it and clears flag...

Review Questions

- How much memory is used by the PRUs?
- What function allows a Linux app to access PRU memory?
- What is a robust way to have the Linux app and PRU code know where values are in shared memory?
- Why do we pad a structure? When should this happen?

Summary

- PRU Memory
  - 8KB per PRU
  - 12KB shared
  - Can use a struct to define which values are in shared memory
- Linux <==> PRU Memory
  - Linux app calls mmap() to request access to PRU memory
- Alignment / Packing
  - PRU byte aligns structs, Cortext A8 word aligns
  - pad structs to line up data