Real-Time Operating System on M4F

CSE4354/5354

C Variable Scope

- Global variables are valid to any function in the program
- Local variables are valid only within a function
- If a variable name is defined globally and locally, the local variable will be referenced unless the local variable is weak

Local Variables

- **Auto (dynamic)**
  - **Each time a function is (re)started**
    - **Variable is dynamically allocated from memory**
    - **Memory can be reused by other functions**
    - **May be optionally initialized**
  - **Usage: *int i = 10; // default usage is auto***
- **Static**
  - **The variable is assigned a unique static location in memory**
  - **Before the function is run the first time (often at program start), the variable may be optionally initialized**
  - **The value is retained until the program ends**
  - **Usage: *static int i = 10;***

Scratch Variables

- Consider v = ((x + y) << 3) | ((w + z) << 1)
- C compiler might need to store w+z or x+y as an intermediate result
- C compiler could store these results in a register (if available) or treat this variable as auto variable and allocate space on the stack to store this result

Auto Variables

- **On machines with data stacks (like the M4F and almost all processors), auto variables are placed on the stack**
- **On some other limited stack/RAM architectures (some simple uP’s have only a call stack) the compiler will assign a memory location that is used for each calling parameter and local variable out of a pool of reusable locations**

Reentrancy (1/2)

- A function is reentered if
  - The function calls itself ("flood fill"/erosion operators)
  - The function is interrupted before completion and it is recalled (an issue for interrupts and preemptive RTOSs)
- When a C function executes, the function is reentrant only if each time the function is called, a separate memory location (usually the stack) is designated to store passing parameters and local variables

Reentrancy (2/2)

- The use of local static, global variables, and heap allocation can create issues if a function if reentered
- Hint: Do not expect a C lib function to be reentrant
  - Be careful with alloc, free, printf, rand, …
  - Look for the reentrant (_R) version of a function if available
- Generally not an issue with cooperative RTOS

RTOS

- **Cooperative**
  - **Relies solely on the tasks to relinquish control**
  - **Many would say this is not a real RTOS solution, but it could be deterministic**
- **Preemptive**
  - **Uses a timer interrupt to distribute time between the tasks**
  - **Tasks may still cooperate and relinquish control**

Common Kernel Functions

- **Yield**
  - Called by a task to voluntarily relinquish time to the kernel
- **Sleep (time_ms)**
  - Called by a task to voluntarily give up all CPU time until at least time_ms has elapsed
- **Wait (semaphore)**
  - Called by a task to determine if a semaphore is available
  - If count > 0, the function immediately returns
  - Else the task is placed in a pending queue and CPU time is given to other application until the semaphore is available
- **Post (semaphore)**
  - Called by a task to indicate that a semaphore is available
  - A task pending execution will be released to continue
  - Optionally, execution can immediately go to the pending task

Additional Kernel Functions

- createTask(pFn, priority)
  - Called to add a task
- destroyTask(pFn)
  - Called to destroy a task and reclaim resources allocated to this task
- rtosRun()
  - Called to start the OS for the first time

Task States

- Ready
  - The task that is unrun, cooperatively yielded time, or was preempted
  - The task is ready to be called again at any time
  - The scheduler will dispatch this function again when the priority is meet again
- Delayed
  - The task that is waiting for some period of time to elapse
- Blocked
  - The task is waiting for a resource to be available
  - The scheduler will dispatch this function again when the priority is meet again and the condition blocking the function is removed
- Running
  - The task is currently running

Task Examples (1)

```
void idle()
{
  while(true)
  {
    PIN_ORANGE = 1;
    waitMicrosecond(1000);
    PIN_ORANGE = 0;
    yield();
  }
}
```

```
void flash_4hz()
{
  while(true)
  {
    PIN_GREEN ^= 1;
    sleep(125);
  }
}
```

Task Examples (2)

```
void one_shot()
{
  while(true)
  {
    wait(flash_req);
    PIN_YELLOW = 1;
    sleep(1000);
    PIN_YELLOW = 0;
  }
}
```

```
void part_of_lengthy_fn()
{
  // represent some lengthy
     operation
  waitMicrosecond(1);
  // give another task a chance
  yield();
}

void lengthy_fn()
{
  long i;
  while(true)
  {
    for (i = 0; i < 4000; i++)
    {
      part_of_lengthy_fn();
    }
    PIN_RED ^= 1;
  }
}
```

Task Examples (3)

```
void read_keys()
{
  int buttons;
  while(true)
  {
    wait(key_released);
    buttons = 0;
    while (buttons == 0)
    {
      buttons = read_pbs();
      yield();
    }
    post(key_pressed);
    if ((buttons & 1) != 0)
    {
      PIN_YELLOW ^= 1;
      PIN_RED = 1;
    }
    if ((buttons & 2) != 0)
    {
      post(flash_req);
      PIN_RED = 0;
    }
    yield();
  }
}
```

```
void debounce()
{
  int count;
  while(true)
  {
    wait(key_pressed);
    count = 10;
    while (count != 0)
    {
      sleep(5);
      if (read_pbs() == 0)
        count--;
      else
        count = 10;
    }
    post(key_released);
  }
}
```

Task Examples (4)

```
void uncooperative()
{
  while(true)
  {
    while (read_pbs() == 8);
    yield();
  }
}
```

```
int read_pbs()
{
  // return number from
  // 0 to N-1;
}
```

Calling Fns Programatically

- When a task is created, the starting address of the task function is saved
- This address can be used to call the task by address by address (a function ptr)
  - typedef void (*_fn)();
  - _fn fn;
  - fn = *start_addr*;
  - (*fn)();

Kernel Fns for Coop RTOS

- The caller must save R0-3, R12
- The callee must save R4-11, SP, LR
- When a task calls yield(), sleep(), wait(), or post(), the callee saved variables must be preserved until program flow returns to the caller
- There are additional requirements when preemption is enabled

ARM Compiler C++ Register Convention

TI SPNU151i manual

Register	Alias	Usage	Preserved by Function[1]
R0	A1	Argument register, return register, expression register	Parent
R1	A2	Argument register, return register, expression register	Parent
R2	A3	Argument register, expression register	Parent
R3	A4	Argument register, expression register	Parent
R4	V1	Expression register	Child
R5	V2	Expression register	Child
R6	V3	Expression register	Child
R7	V4, AP	Expression register, argument pointer	Child
R8	V5	Expression register	Child
R9	V6	Expression register	Child
R10	V7	Expression register	Child
R11	V8	Expression register	Child
R12	V9, 1P	Expression register, instruction pointer	Parent
R13	SP	Stack pointer	Child[2]
R14	LR	Link register, expression register	Child
R15	PC	Program counter	N/A
CPSR		Current program status register	Child
SPSR		Saved program status register	Child

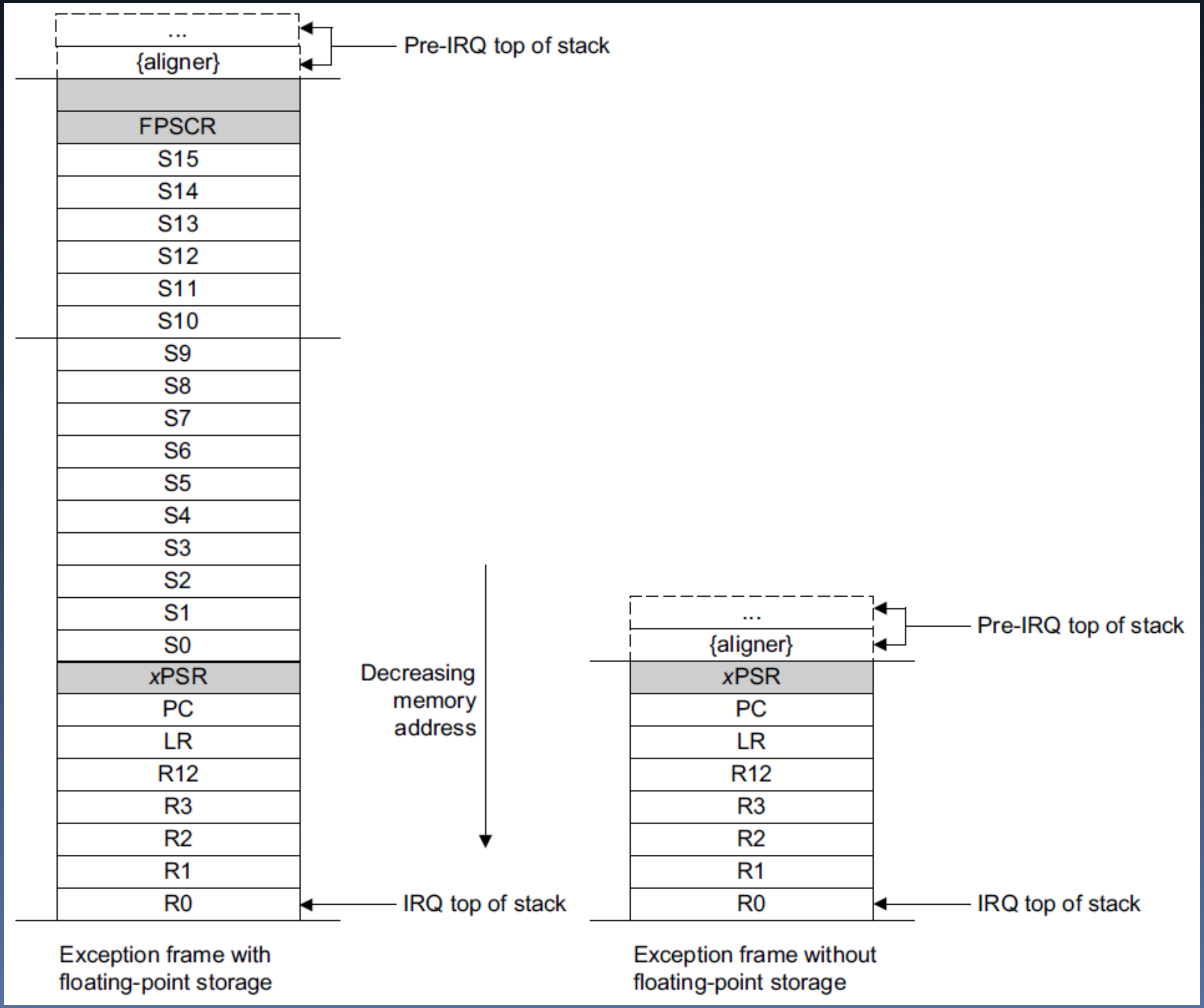
(1) The parent function refers to the function making the function call. The child function refers to the function being called.

(2) The SP is preserved by the convention that everything pushed on the stack is popped off before returning.

M4F Stack Operation

- **PUSH *reglist* pushes the registers to memory with the lowest register number being placed in the lowest memory location**
- **POP *reglist* reverses this process**
- **SP decrements with PUSH and increments with POP**
- **After an exception occurs, a large number of registers are pushed on the stack (see next slide)**
- **Exception result is placed in the LR register**

M4F Stack Operation



Exception Return Type

ARM DUI0553A manual

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

Preemption on M4F

- In addition to the cooperative kernel functions, a preemptive RTOS uses a timer interrupt to interrupt tasks and switch the context to another task
- The context saved/restored automatically during interrupt call/return is R0-3, R12, xPSR, LR, PC, and optionally the FP stack
- The user must manually save all other registers in addition to these values

Kernel Fn Changes for Preemption

- The following kernel functions must be modified
  - yield, sleep, and wait
  - post (if optional context switching is performed)
- Must support a task that cooperatively relinquished operation (yield, sleep, wait, and perhaps post) which begins running later via preemption and the converse
- All kernel functionality should all be realized in the SysTickIsr(), SvCallIsr(), and PendSvIsr() functions