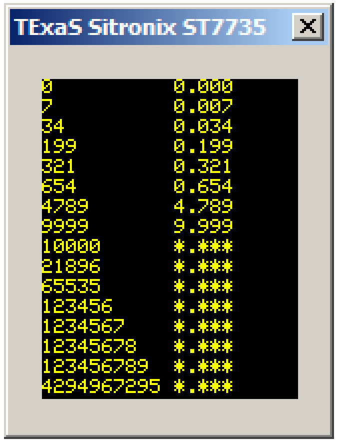
COMP462: Embedded Systems

Lecture 9: Stack and Local Variables, Fixed-Point Numbers, Busy-wait and LCD

Agenda

- ❑Local Variables
- ❑Stack and Activation Records
  - ❖Register allocation
  - ❖SP-relative addressing
  - ❖R11-relative addressing (stack frame pointer)
- ❑Fixed-point numbers
- ❑LCD Interfacing



Local Variables

- **Scope**
  => from where can it be accessed?
  - **Private** means restricted to
    - function where it is defined
    - specific code block { ... }
    - file where is it defined
  - **Public** means any software can access it
- **Allocation/Lifetime/Persistence**
  => when is it created & destroyed?
  - **Dynamic** allocation using registers or stack
  - **Permanent** allocation assigned in RAM

Local Variables

- **Local or automatic variables**
  => Private scope, dynamic allocation
  - Temporary information
  - Scope: used only inside the function
  - Lifetime
    - Allocated when entered,
    - Used in the body, then
    - Deallocated on exit
  - Implementation
    - Registers
    - Allocate on stack and use SP to access
    - Allocate on stack and use R11 (stack frame pointer)

Variables in C

❑ Global

- ❖ Public scope
- ❖ Permanent allocation
- ❖ Bad style

```
// accessible by all modules
int16_t myGlobalVariable;
```

❑ File-private

- ❖ Private scope to file
- ❖ Permanent allocation
- ❖ Sharing: ISR ⇔ Functions

```
//accessible this file only
static int16_t myPrivateStaticVariable;
```

❑ Local - Automatic

- ❖ Private scope,
- ❖ Dynamic allocation

```
void MyFunction(void){
  int16_t myLocalVariable;
}
```

❑ Local - Static

- ❖ Private scope to function
- ❖ Permanent allocation

```
void MyFunction(void){
   static int16_t count=0;
   count++;
}
```

Reduce the scope as much as possible: need to know basis

Functions in C

- Public Function
  - Place a prototype in header file
  - Module specified in name

```
// callable by all modules
void SysTick_Init(void){…}
```

- Private Function
  - No prototype in header file

```
// callable by other
// routines in this file only
void static MyPrivateFunction(void){…}
```

Why use Locals?

- Allocation/release allows reuse of memory
- Limited scope provides for data protection
- Only program that created it can access it

Why use Stack?

- Large number
- Arrays

Why use Registers?

- Simple
- Fast

Recall Stack Rules

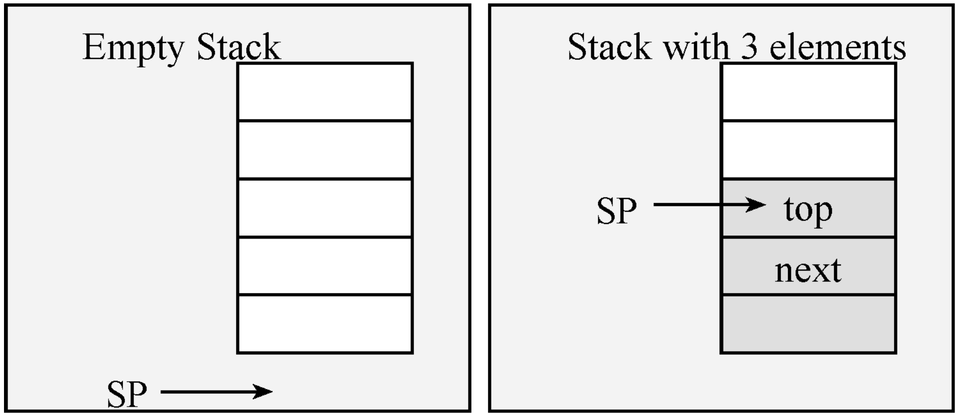
- Program segments should have an equal number of pushes and pulls
- **Push** with multiple registers will always put the lower numbered register's contents in the lower address.
- **Pop** with multiple registers will always get the lower numbered register's contents from the lower address.

Push

1. **SP=SP-4**
2. **Store 32 bits at SP**

Pop

1. **Read 32 bits at SP**
2. **SP=SP+4**



Variables on the stack

- Many inputs or arrays

```
void play_note(uint16_b pitch,
               uint16_b duration,
               uint16_b loud_left,
               uint16_b loud_right,
               uint16_b timber_idx,
               uint16_b attack_rate,
               uint16_b attack_type,
               uint16_b decay_rate,
               uint16_b decay_type);

void Function(void){char Buffer[100];
}
```

Local variables using Registers

```
; *****binding phase***********
sum  RN   4  ;32-bit unsigned
n    RN   5  ;32-bit unsigned
; 1)**** no allocation phase **
Calc PUSH {R4,R5}
     MOV  n,R0
; 2)******access phase ********
     MOV  sum,#0
loop ADD  sum,sum,n   ;sum+n
     SUBS n,n,#1      ;n-1
     BNE  loop
; 3)***no deallocation phase **
     MOV  R0,sum
     POP  {R4,R5}
     BX   LR          ;R0=sum
```

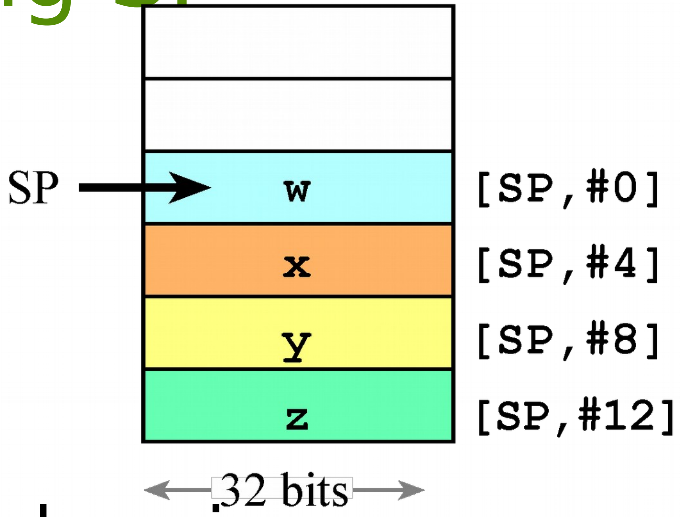
```
// input: n 32-bit number
// output: n+(n-1)+(n-2)+...+2+1
uint32_t Calc(uint32_t n) {
  uint32_t sum;
  sum = 0;
  do{
    sum=sum+n;
    n--;
  } while(n>0);
  return sum;
}
```

R4	sum
R5	n

Use R4-R11 as locals when this function calls another, because the other function by AAPCS will preserve the values of R4-R11

Use R0-R3,R12 as locals when this function doesn't call another function, because you do not need to preserve R0-R3,R12

Stack frame using SP



- **Binding** using EQU
  - Draw a stack figure

```
y    EQU 8
```

- **Allocate** by making space

```
SUB SP,SP,#16
```

- **Access** using SP-relative addressing

```
STR R0,[SP,#y]
```

- **Deallocate** by freeing (balancing) stack

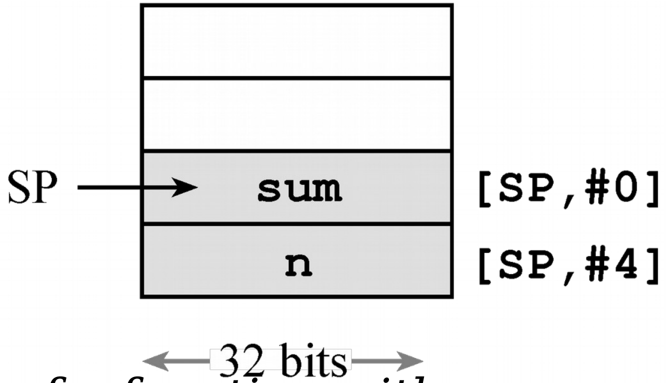
```
ADD SP,SP,#16
```

Stack frame using SP

```
; *****binding phase***************
sum   EQU  0  ;32-bit unsigned number
n     EQU  4  ;32-bit unsigned number
; 1)*****allocation phase *********
Calc PUSH {R0}   ;allocate, init n
     SUB  SP,#4  ;allocate sum
; 2)******access phase ************
     MOV  R0,#0
     STR  R0,[SP,#sum] ;sum=0
loop LDR  R1,[SP,#n]   ;R1=n
     LDR  R0,[SP,#sum] ;R0=sum
     ADD  R0,R0,R1     ;R0=sum+n
     STR  R0,[SP,sum]  ;sum=sum+n

     LDR  R1,[SP,#n]   ;R1=n
     SUBS R1,R1,#1     ;n-1
     STR  R1,[SP,#n]   ;n=n-1
     BNE  loop
; 3)******deallocation phase *****
     LDR  R0,[SP,#sum] ;R0=sum
     ADD  SP,#8     ;deallocation
     BX   LR        ;R0=sum
```

```
// input: n 32-bit number
// output: n+(n-1)+(n-2)+...+2+1
uint32_t Calc(uint32_t n) {
  uint32_t sum;
  sum = 0;
  do{
    sum=sum+n;
    n--;
  } while(n>0);
  return sum;
}
```

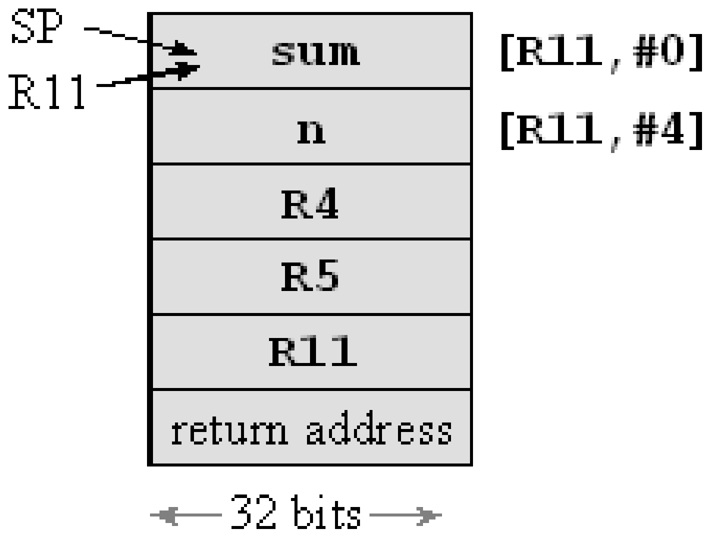


Stack pointer implementation of a function with two local 32-bit variables.

Frame pointer using R11

```
; *****binding phase***************
sum  EQU  0  ;32-bit unsigned number
n    EQU  4  ;32-bit unsigned number
; 1)*****allocation phase *********
calc PUSH {R4,R5,R11,LR}
     SUB  SP,#8  ;allocate n,sum
     MOV  R11,SP ;frame pointer
; 2)******access phase ************
     MOV  R0,#0
     STR  R0,[R11,#sum] ;sum=0
     MOV  R1,#1000
     STR  R1,[R11,#n]   ;n=1000
loop LDR  R1,[R11,#n]   ;R1=n
     LDR  R0,[R11,#sum] ;R0=sum
     ADD  R0,R1         ;R0=sum+n
     STR  R0,[R11,sum]  ;sum=sum+n
     LDR  R1,[R11,#n]   ;R1=n
     SUBS R1,#1         ;n-1
     STR  R1,[R11,#n]   ;n=n-1
     BNE  loop
; 3)******deallocation phase *****
     ADD  SP,#8    ;deallocation
     POP  {R4,R5,R11,PC}   ;R0=sum
```

```
uint32_t calc(void){
uint32_t sum,n;
  sum = 0;
  for(n=1000;n>0;n--){
    sum=sum+n;
  }
  return sum;
}
```

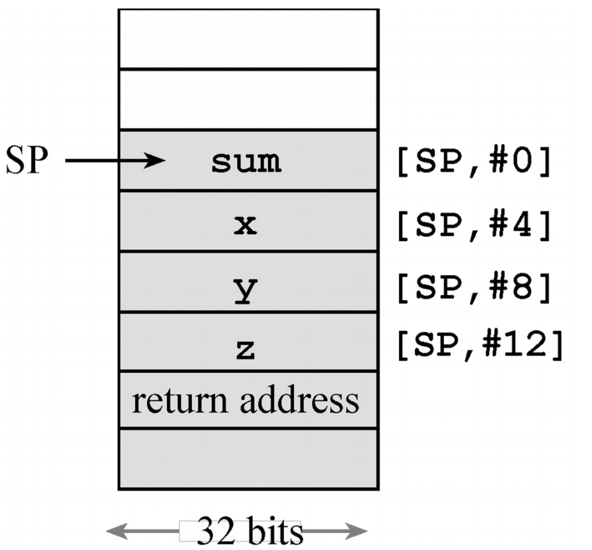


Frame pointer implementation of a function with two local 32-bit variables.

Push parameters on stack

```
; Inputs R0 is x
;        R1 is y
;        R2 is z
; Output R0 is return value
sum  EQU  0  ;32-bit signed number
x    EQU  4  ;32-bit signed number
y    EQU  8  ;32-bit signed number
z    EQU  12 ;32-bit signed number
Add3 PUSH {R0,R1,R2,LR}
     SUB  SP,#4  ;allocate sum
; body of the function
     LDR  R0,[SP,#x]
     ADD  R0,R0,[SP,#y]
     ADD  R0,R0,[SP,#z]
     STR  R0,[SP,#sum]
     ADD  SP,#16 ;deallocate
     POP  {PC}
```

```
int32_t Add3(int32_t x, int32_t y,
             int32_t z) {
  int32_t sum;
  sum = x+y+z;
  return sum;
}
```



Pushing parameters on stack makes them similar to local variables

Fixed-Point Revisited

- **Why:**
  express non-integer values
  no floating point hardware support (want it to run fast)
- **When:**
  range of values is known
  range of values is small
- **How:**
  1) **variable integer**, called **I**.
     may be signed or unsigned
     may be 8, 16 or 32 bits (**precision**)
  2) **fixed constant**, called **Δ (resolution)**
     value is fixed, and can not be changed
     not stored in memory
     specify this fixed content using comments

value ≡ integer • **Δ**

Fixed-Point Numbers: Decimal

Decimal Fixed-Point

(Value = $I*10^m$)

I is a 16-bit unsigned integer (variable integer)

$\Delta = 10^m$ decimal fixed-point (fixed constant)

For example with m=-3 (resolution of 0.001 or milli) the value range is 0.000 to 65.535 (with 16-bit)

What is π represented as, in Decimal Fixed-Point?

π (3.14159…) = $I*10^{-3}$

=> I = Integer approximation of ($3.14159…*10^3$)

I = Integer approximation of (3141.59)

I = 3142

Decimal Fixed-Point numbers are human-friendly

-easy to input/output to humans

Fixed-Point Numbers: Binary

Binary Fixed-Point

(Value = $I*2^m$)

I is a 16-bit unsigned integer (variable integer)

$\Delta = 2^m$ binary fixed-point (fixed constant)

For example with m=-8 (resolution of 1/256)

What is π represented as, in binary Fixed Point?

π (3.14159…)= $I*2^{-8}$

=> I = Integer approximation of($3.14159\ldots*2^8$)

I = Integer approximation of(804.2477)

I = 804

Binary Fixed-Point numbers are computer-friendly

-runs very fast because shifting is fast

Fixed-Point Math Example

Consider the following calculation.

$$C = 2 * \pi * R$$

The variables C, and R are integers

$2\pi \approx 6.283$

$$C = (6283 * R)/1000$$

Fixed-Point Math Example

Calculate the volume of a cylinder

$$V = \pi * R^2 * L$$

The variables are fixed-point

$R = I*2^{-4}$ cm $\quad L = J*2^{-4}$ cm

$V = K*2^{-8}$ cm^3 $\quad \pi \approx 100*2^{-5}$

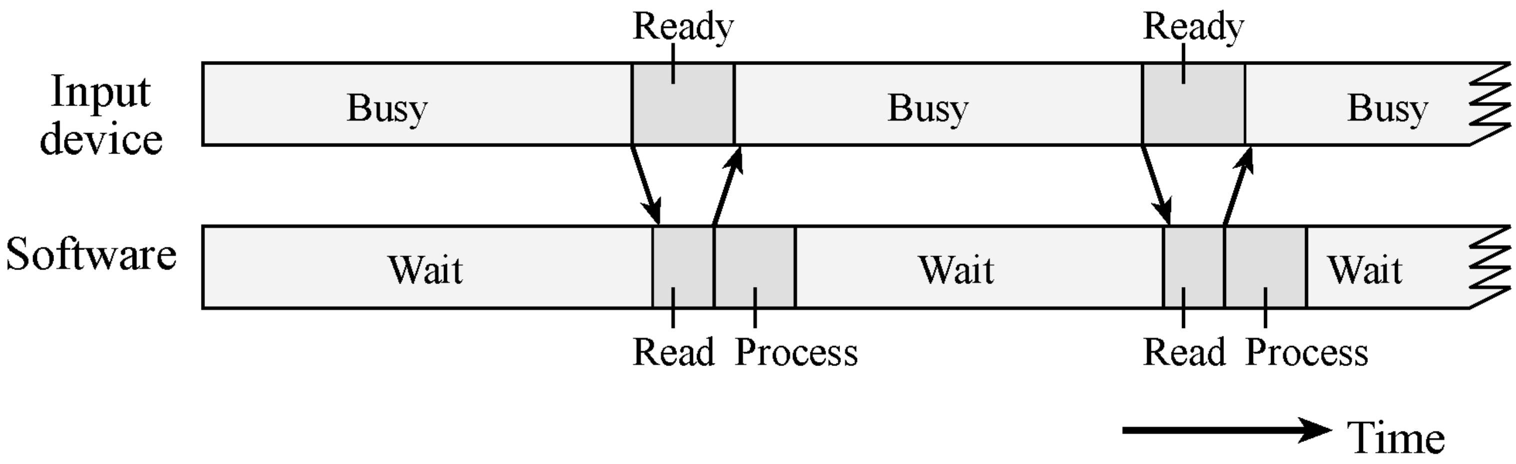
$$K = (100*I*I*J)>>9$$

Input/Output Synchronization

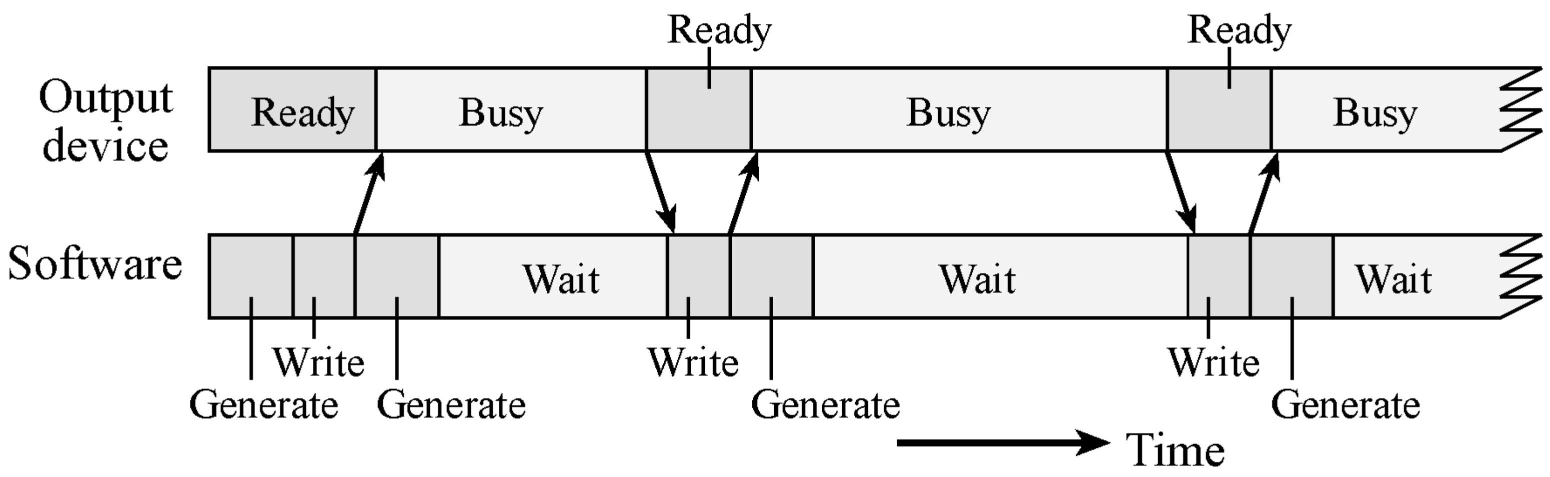
- Processor-Peripheral Timing Mismatch
  - Peripherals, *e.g.*, displays, sensors, switches, generally operate MUCH slower than processor instruction times
    - Processor ~ MHz
    - Peripheral ~ kHz or Hz
  - MANY instructions can be executed while peripheral processes information

Input/Output Sync. (cont.)

INPUT



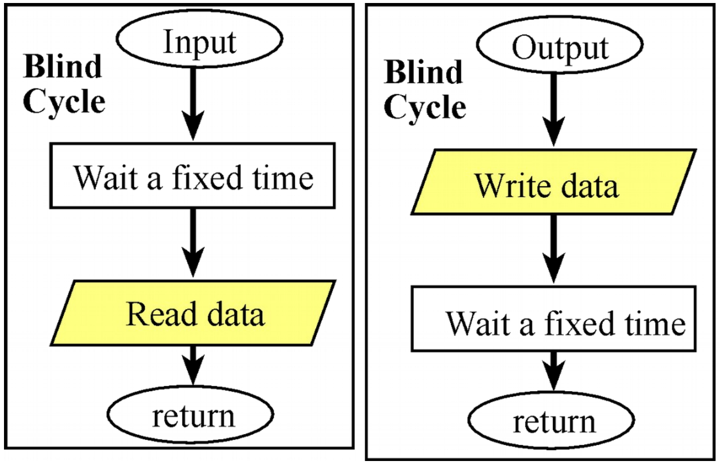
OUTPUT



I/O Sync Options (1)

What to do while the peripheral is BUSY?

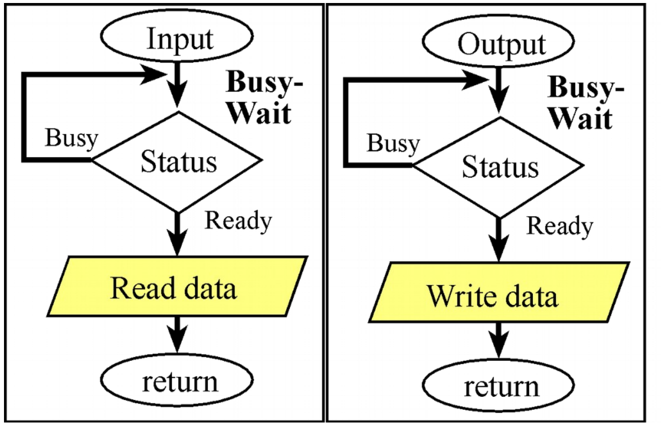
1. BLIND CYCLE TRANSFER
   - Suppose that a BUSY control signal is not available
   - Perform I/O operation
   - Wait for a period of time that is guaranteed to be sufficient for operation to complete
   - Initiate next operation



I/O Sync Options (2)

What to do while the peripheral is BUSY?

2. BUSY-WAIT (e.g., ready-busy, test-transfer)
   - Poll peripheral status – wait for READY/NOT BUSY
   - Perform other tasks between polls
   - Unless timed correctly, *under/over run* possible
     - One solution: POLL CONTINUOUSLY

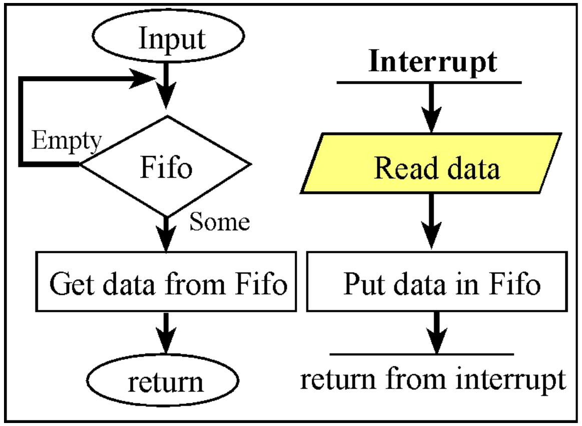


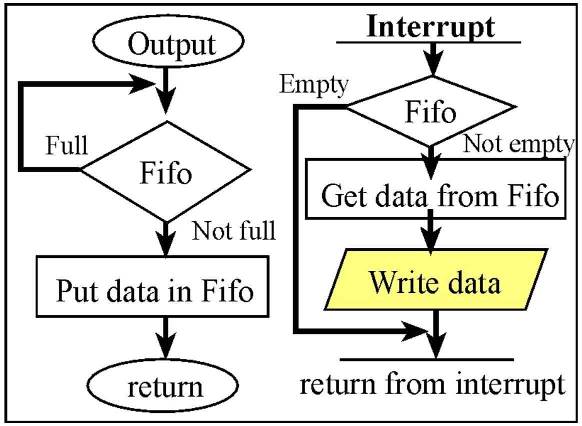
I/O Sync Options (3)

What to do while the peripheral is BUSY?

3. INTERRUPT/TRANSFER

- Hardware INTERRUPTS processor on condition of READY/NOT BUSY
- Facilitates performing other – *background* - processing between I/O transfers
  - Processor changes context when current transfer complete
  - Requires program structure to process context change



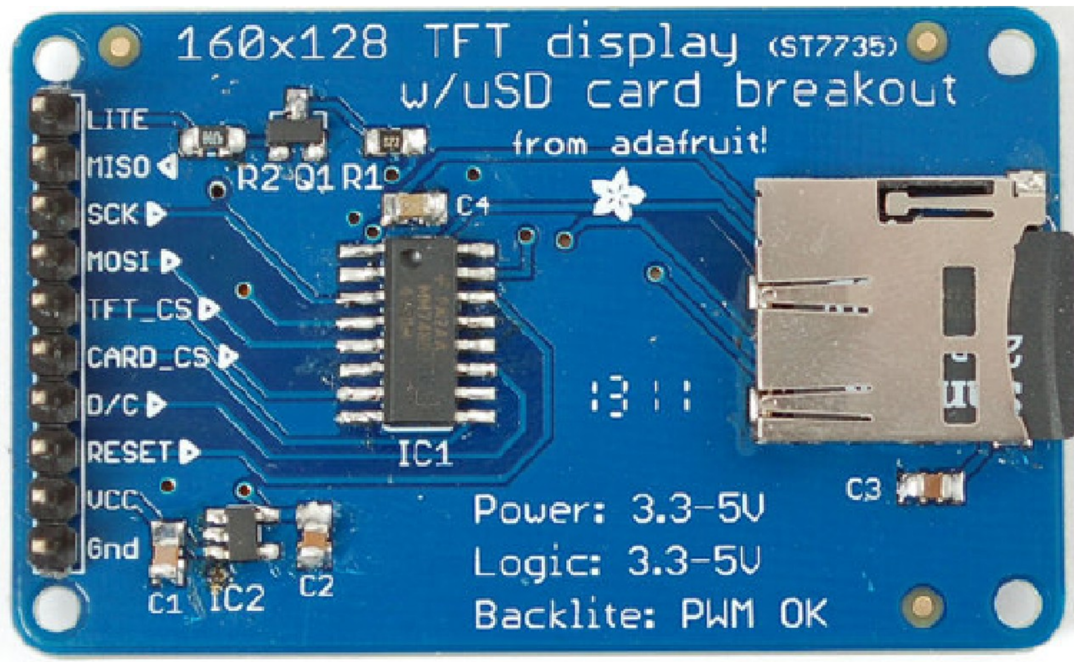


Sitronix ST7735 LCD

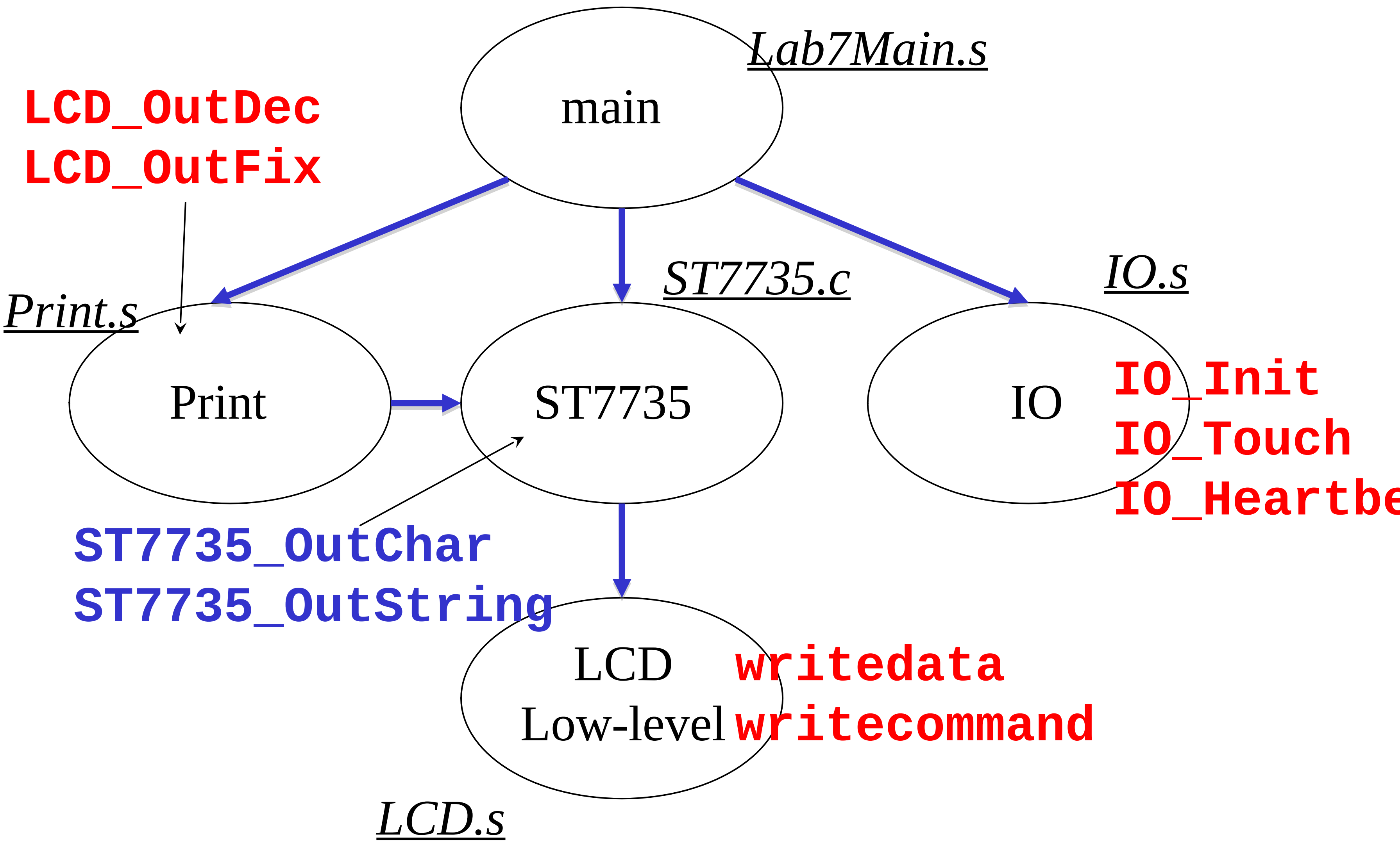
- ❑ Resolution: WxD of 128x160 pixels
- ❑ 1.8” TFT LCD display with 18-bits per pixel
- ❑ On-chip Display Data RAM 128x160x18bits
- ❑ Device driver library ST7735.c provided to you – Implements the SPI protocol
- ❑ Interfaced using the SPI protocol with 4 or 5 wires
- ❑ Built-in micro-SD card for storage

Interface to Launchpad

```
// pin 10 Backlight    +3.3 V
// pin 9  MISO         unconnected
// pin 8  SCK          PA2 (SSI0Clk)
// pin 7  MOSI         PA5 (SSI0Tx)
// pin 6  TFT_CS       PA3 (SSI0Fss)
// pin 5  CARD_CS      unconnected
// pin 4  D/C          PA6 (GPIO)
// pin 3  RESET        PA7 (GPIO)
// pin 2  VCC          +3.3 V
// pin 1  Gnd          ground
```



Module Call Graph



LCD Programming

writecommand: Involves 6 steps performed to send 8-bit Commands to the LCD

1. Read SSI0_SR_R and check bit 4,
2. If bit 4 is high, loop back to step 1
   - wait for BUSY bit to be low
3. Clear D/C=PA6 to zero
   - (D/C pin configured for COMMAND)
4. Write the command to SSI0_DR_R
5. Read SSI0_SR_R and check bit 4,
6. If bit 4 is high loop back to step 5
   - (wait for BUSY bit to be low)

Think about what happens when you output multiple commands one right after another?

LCD Programming

writedata: Involves 4 steps performed to send 8-bit Commands to the LCD:

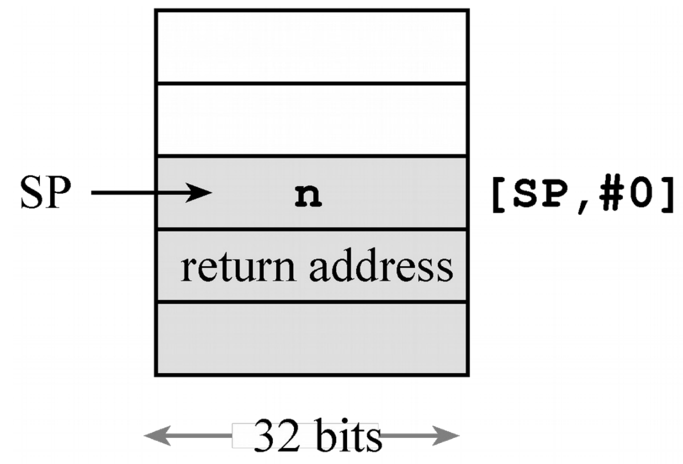
1. Read SSI0_SR_R and check bit 1,
2. If bit 1 is low, loop back to step 1
   - (wait for TNF bit to be one)
3. Set D/C=PA6 to one
   - (D/C pin configured for DATA)
4. Write the 8-bit data to SSI0_DR_R

Think about what happens when you output multiple data one right after another?

Recursion using the stack (skip)

```
; Input  R0 is n
; Output R0 is return value
n    EQU  0  ;input parameter
Fact PUSH {R0,LR}
     CMP  R0,#1
     BLS  base
     SUB  R0,#1  ;n-1
     BL   Fact   ;Fact(n-1)
     LDR  R1,[SP,#n]
     MUL  R0,R0,R1 ;n*Fact(n-1)
     B    done
base MOV  R0,#1
done ADD  SP,#4 ;deallocate
     POP  {PC}
```

```
uint32_t Fact(uint32_t n) {
  if(n<=1) return 1;
  return n*fact(n-1);
}
```



Recursion requires putting parameters and locals on the stack

I/O Sync Options (4)

What to do while the peripheral is BUSY?

4. DIRECT MEMORY ACCESS TRANSFER
   - Special purpose hardware logic monitors status of BUSY signal and maintains addresses of data to be communicated
     - ➢ Requires address and block size initialization
   - On the condition of NOT BUSY logic communicates next data element and increments address
   - When transfer is complete, logic provides COMPLETE INTERRUPT

Our TM4C123 supports DMA (but EE319K doesn't use it)

Interface to Launchpad

Sitronix ST7735	**LaunchPad Pin**
10 – Backlight	Power (3.3V)
9 – MISO	Not Connected
8 – SCK	PA2 (SSI0Clk)
7 – MOSI	PA5 (SSI0Tx)
6 – TFT_CS	PA3 (SSI0Fss)
5 – CARD_CS	Not Connected
4 – Data/Command	PA6 (GPIO: High for Data Low for Command)
3 – RESET	PA7
2 – Vcc	Power (3.3v)
1 – Gnd	Ground

SPI pins