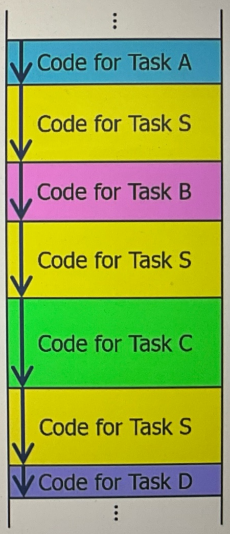
Subroutines

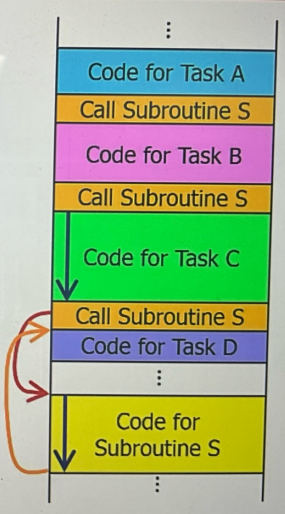
Repeated Tasks

- Sometimes a task must be performed multiple times within a program
  - Could repeat the code multiple times in the program...
- Downsides:
  - Multiple copies to test/debug
  - Multiple copies to fix if a bug is found
  - Multiple copies occupying memory
  - ...and more!



Subroutines

- Could convert the repeated task into a **subroutine**
  - One copy of the code that is **called** from multiple places within the program
  - After it executes, it **returns** to the code just after the call



Subroutines

- Subroutines are well-defined software modules that are written to perform a specific task
  - Functions and methods in high-level languages are implemented using subroutines
- To use a subroutine, a program **calls** the subroutine
  - The processor executes the subroutine code
  - When the subroutine code is finished, the processor **returns** to the next instruction after the one that called the subroutine
- A program can call a subroutine multiple times
  - Subroutines can also call other subroutines

(Subroutines are functions/methods in high-level languages)

Why use Subroutines?

- Write code that performs a task, then call that code multiple times in a program
  - Only one copy of the code needs to exist
- More abstract approach to programming by **encapsulating** lower-level tasks
  - Code calling the subroutine does not need to know the details of how the subroutine code actually works
- Write **modular** code to simplify testing and increase possibilities for software reuse
  - Testing a small subroutine is much simpler than testing an entire program
  - Creating a library of known-good software modules allows higher-level code to more easily use them

Subroutine Structure

- A subroutine needs a label (a name representing its starting address) and a RETurn instruction

```
; Name: abs
; Purpose: computes absolute value of an operand
abs                          ; Label

        ; we'll need to add some code here

Return  RET                  ; return to caller
```

- A subroutine must ALWAYS return execution to the instruction after the one that called it
  - **Every** subroutine will end with the RET instruction
- Should also include a comment header to explain what the subroutine does and how to use it

Calling a Subroutine

(★ Note: NO BR/JMP in subroutines)

- A subroutine is called using the JSR (**J**ump to **S**ub**R**outine) instruction with the subroutine label
  - Syntax: `JSR label`
- ☆ The JSR instruction does two things
  - R7 ← already-incremented PC
  - PC ← address of label
- The next instruction fetched will be the first instruction of the subroutine
- R7 contains the address of the instruction after the JSR instruction
  - This is how the subroutine will be able to return execution to the instruction after the calling instruction

① JSR copies the PC+1 to R7
② JSR brings the current PC to subroutine
③ After subroutine execution, go back to the PC+1 stored in R7

Returning from a Subroutine

- To return to the caller, the RET instruction copies R7 into the PC
  - R7 contains the address of the instruction after the JSR, so execution returns to the caller code
  - Note that RET is actually an alias for JMP R7 – but you should always use RET to return from a subroutine!
- If a subroutine calls another subroutine (or otherwise modifies R7), it MUST save a copy of R7 to memory before calling the subroutine, and then restore R7 before its RET instruction

Programming tip: Try not use R7 in your code unless you really need to. Accidentally corrupting R7 leads to "interesting" behavior that can be hard to debug!

Parameter Passing

- Many subroutines require information from the caller to perform their task, and may need to return one or more results to the caller
  - Commonly called **parameters**
  - For example in **abs**, where does the caller put the value to convert, and where the result should be placed?
- Parameters can be passed to and from the subroutine in a variety of ways
  - In registers → By value / By reference
  - In specific memory locations
  - On the stack
    - A special data structure implemented in most processors

1

Pass-By-Value

- In **abs**, we dictate the operand must be passed-by-value in R0, and the result returned in R0

```
; Name: abs
; Purpose: computes absolute value of an operand
; Assumes: R0 = operand value
; Returns: R0 = result
abs
        ADD R0, R0, #0   ; check sign of R0
        BRzp abs_exit    ; no change if pos or zero
        NOT R0, R0       ; was negative, so negate!
        ADD R0, R0 #1
abs_exit
        RET              ; return to caller
```

Programming tip: Prefix all subroutine labels with the subroutine name to avoid label duplication/conflicts!

2

Pass-By-Reference

- Pass-by-value is limited by the size and number of registers
  - What if we want to pass a string to a subroutine?
- Pass-by-reference is implemented by passing the **address** of an operand that is stored in memory

```
; Name: strend
; Purpose: return the address of the null terminator
; Assumes: R1 = address of start of ASCIIZ string
; Returns: R1 = address of null terminator
strend
        LDR R0, R1, #0   ; get character
        BRz strend_exit  ; return if terminator
        ADD R1, R1, #1   ; increment string pointer
        BR  strend
strend_exit
        RET              ; return to caller
```

Register Usage

- Note that **strend** uses register (R0) to hold the character loaded by the LDR
- But what if the caller is also using R0?
  - The subroutine call will corrupt it!

Conflicts!

```
; Name: strend
; Purpose: return the address of the null terminator
; Assumes: R1 = address of start of ASCIIZ string
; Returns: R1 = address of null terminator
strend
      LDR R0, R1, #0  ; get character
      BRz strend_exit ; return if terminator
      ADD R1, R1, #1  ; increment string pointer
      BR  strend
strend_exit
      RET             ; return to caller
```

Preventing register corruption

- **Caller-save:** the caller code is responsible for saving/restoring any needed register values
  - Store the values in those registers to memory before calling the subroutine
  - Re-load the saved values after the subroutine returns
- **Callee-save:** the subroutine is responsible for saving/restoring any modified register values
  - Subroutine must appear to not modify registers…
  - …except for those used to hold results
  - …and the register that holds the return address (R7)!

Context save and restore

Fixing strend

- Adding callee-save code to save/restore R0

```
; Name: strend
; Purpose: return the address of the null terminator
; Assumes: R1 = address of start of ASCIIZ string
; Returns: R1 = address of null terminator
strend
      ST  R0, strend_R0 ; context save
strend_loop
      LDR R0, R1, #0    ; get character
      BRz strend_exit   ; return if terminator
      ADD R1, R1, #1    ; increment string pointer
      BR  strend _loop
strend_exit
      LD  R0, strend_R0 ; context restore
      RET               ; return to caller
strend_R0 .BLKW 1
```

Example Context Save and Restore

- **mul5** saves/restores register R1
  - Callee-save
- **mul25** saves/restores R7 so that **mul25** can return correctly
  - Caller-save
  - R7 is changed by `JSR mul5` instruction, so cannot be saved within **mul5**

```
; Name: mul5
; Purpose: multiplies operand by 5
; Assumes: R0 = operand value
; Returns: R0 = result
mul5
  ST R1, mul5_R1 ; context save
  ADD R1, R0, R0 ; R1 is 2*R0
  ADD R1, R1, R1 ; R1 is 4*R0
  ADD R0, R0, R1 ; R0 is now 5*R0
  LD R1, mul5_R1 ; context restore
  RET            ; return to caller
mul5_R1 .BLKW 1
```

```
; Name: mul25
; Purpose: multiplies operand by 25
; Assumes: R0 = operand value
; Returns: R0 = result
mul25
  ST R7, mul25_R7 ; context save
  JSR mul5
  JSR mul5        ; R0 = 5*5*R0
  LD R7, mul25_R7 ; context restore
  RET             ; return to caller
mul25_R7 .BLKW 1
```

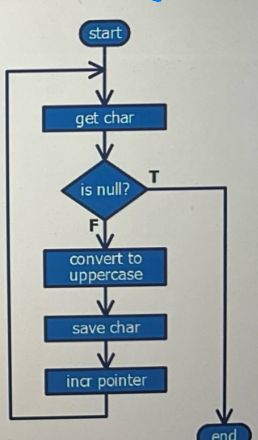
Programming With Subroutines

Program Design and Coding

- Top-down design
  - Start the design at the top level, then work our down through the subroutine hierarchy
  - Develop flowcharts for each subroutine
- Bottom-up implementation
  - Once we have all the subroutines designed, we will actually code them starting from the lowest level
  - Each subroutine is thoroughly tested before we move up the hierarchy

strupr Design

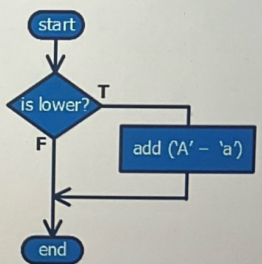
- Write a subroutine **strupr** that converts every lowercase letter in the string to uppercase
  - Subroutine is passed the address of an ASCIIZ string in R1
    - Uses pass-by-reference
  - Does not return a result
- Assume that we will create a subroutine **toupper** to perform the work on each character



toupper deals with single character

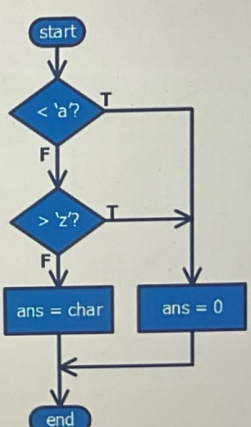
`toupper` Design

- Write a subroutine **toupper** that converts a single character to uppercase
  - Takes a single input in register R0, returns result in R0
  - Only modifies R0 if it contains the ASCII code for a lowercase letter
- Assume that we will create a subroutine **islower** to perform the test for lowercase



`islower` Design

- Write a subroutine **islower** that tests if a single character is lowercase
  - Takes a single input in register R1, returns result in R1
  - If R1 contains ASCII code for a lowercase letter, returns a non-zero value; otherwise returns 0
    - Can just leave R1 as-is if lowercase because ASCII code is already non-zero



```
; Determines if a character is lowercase
; Assumes: R1 = character to check
; Returns: R1 = orig. character if lowercase, 0 otherwise
islower
    ST    R0, islower_R0      ; context save
    LD    R0, islower_nega    ; is char < 'a'?
    ADD   R0, R1, R0          ; R0 <- char - 'a'
    BRn   islower_false       ; if so, will return 0
    LD    R0, islower_negz    ; is char <= 'z'?
    ADD   R0, R1, R0          ; R0 <- char - 'z'
    BRnz  islower_exit        ; if so, will just return char
islower_false
    AND   R1, R1, #0          ; not lowercase, so return 0
islower_exit
    LD    R0, islower_R0      ; context restore
    RET

islower_R0      .BLKW  1
islower_nega    .FILL  xFF9F ; negative ASCII 'a'
islower_negz    .FILL  xFF86 ; negative ASCII 'z'
```

islower subroutine

islower Testing

- Always test with values that are likely to expose errors in the software's logic
  - In this case, at a minimum need to test with values on both sides of 'a' and 'z'

```
Start
    LD    R1, ch_aminus1 ;test 'a' - 1
    JSR   islower
    LD    R1, ch_a       ;test 'a'
    JSR   islower
    LD    R1, ch_b       ;test 'b'
    JSR   islower
    LD    R1, ch_y       ;test 'y'
    JSR   islower
    LD    R1, ch_z       ;test 'z'
    JSR   islower
    LD    R1, ch_zplus1  ;test 'z' + 1
    JSR   islower
    BRnzp Start
ch_aminus1 .FILL x60
ch_a       .FILL x61
ch_b       .FILL x62
ch_y       .FILL x79
ch_z       .FILL x7A
ch_zplus1  .FILL x7B
```

```
; If a character is lowercase, converts it to uppercase
;      (otherwise, returns the unmodified character)
; Assumes: R0 = character
; Returns: R0 = possibly modified character
toupper
    ST    R7, toupper_R7      ; context save
    ST    R1, toupper_R1
    ADD   R1, R0, #0          ; copy character to R1
    JSR   islower             ; call islower
    ADD   R1, R1, #0          ; test return value
    BRz   toupper_exit        ; if not lower, don't convert
    LD    R1, toupper_a_A     ; otherwise convert to uppercase
    ADD   R0, R0, R1          ;    by adding ('A'-'a')
toupper_exit
    LD    R1, toupper_R1      ; context restore
    LD    R7, toupper_R7
    RET

toupper_R1    .BLKW  1
toupper_R7    .BLKW  1
toupper_a_A   .FILL  xFFE0    ; ASCII 'A'- ASCII 'a'
```

toupper subroutine

toupper Testing

- We know **islower** works, so only testing for proper conversion of lowercase chars, and that others are not changed

```
Start
    LD    R0, lower_a     ;test 'a'
    JSR   toupper
    LD    R0, lower_z     ;test 'z'
    JSR   toupper
    LD    R0, upper_A     ;test 'A'
    JSR   toupper
    LD    R0, upper_Z     ;test 'Z'
    JSR   toupper
    LD    R0, digit_1     ;test '1'
    JSR   toupper

    BRnzp Start

lower_a   .FILL x61
lower_z   .FILL x7A
upper_A   .FILL x41
upper_Z   .FILL x5A
digit_1   .FILL x31
```

```
; Converts all lowercase characters in string to uppercase
; Assumes: R1 = pointer to start of ASCIIZ string
; Returns: Nothing
strupr
    ST    R7, strupr_R7      ; context save
    ST    R1, strupr_R1
    ST    R0, strupr_R0
strupr_loop
    LDR   R0, R1, #0         ; get character
    BRz   strupr_exit        ; if at terminator, we are done
    JSR   toupper            ; otherwise, process the character
    STR   R0, R1, #0         ; write it back to string
    ADD   R1, R1, #1         ; increment pointer (-> next char)
    BRnzp strupr_loop
strupr_exit
    LD    R0, strupr_R0      ; context restore
    LD    R1, strupr_R1
    LD    R7, strupr_R7
    RET

strupr_R0    .BLKW  1
strupr_R1    .BLKW  1
strupr_R7    .BLKW  1
```

(We are not returning R1, that's why we need to context save/restore)

strupr subroutine

strupper Testing

- ALWAYS test string processing subroutines with an empty string
- Need to test enough strings to verify that all characters are processed correctly
  - Always check first and last character
  - Make sure it does not go past the end of the string!

```
Start
    LEA   R1, S1 ;test empty string
    JSR   strupr
    LEA   R1, S2 ;l.c. at ends
    JSR   strupr
    LEA   R1, S3 ;non-l.c. at ends
    JSR   strupr

    BRnzp Start

S1        .STRINGZ   ""
S2        .STRINGZ   "abcABC123abc"
S3        .STRINGZ   "`az{ AZ"
```