Subroutines

- procedures and functions in high level languages are modeled on subroutines
- typically, assembly code is very modular with the main routine less than 100 lines long

Today’s objective: introduction to really simple subroutines to simplify program structure for I/O

- use JSR, *jump to subroutine*, to call subroutine
- use RTS, *return from subroutine*, to return from subroutine
Subroutines

ORG $1000
MAIN ....
LEA msg1,A1 ; point to string
JSR PRT_S ; jump to subroutine PRT_S
...
LEA msg2,A1 ; point to string
JSR PRT_S ; jump to subroutine PRT_S
...
STOP #$2700

PRT_S ...
...
RTS ; return from subroutine

msg1 ...
msg2 ...
END MAIN

...sample structure

← main program

← subroutine PRT_S

← data area
Subroutines

... calling the subroutine

Addressing Range

1. jump to subroutine  
   JSR <ea>  
   e.g. JSR MAXIMUM  
   → normally code using JSR*

2. branch to subroutine  
   BSR <ea>  
   e.g. BSR MINIMUM  
   → smaller machine code  
   BSR = BSR.W  
   BSR.S = BSR.B

*once program complete, can use BSR to tighten code
Subroutines

PC = program counter
→ contains the address of the next instruction to be executed
→ how the processor tracks where you are in the program (*simplified*):
  → processor sends address in PC to memory
  → while memory is retrieving the instruction, PC is incremented to point to next instruction
  → memory returns the instruction
  → processor executes the instruction
Subroutines ... key registers

SP = stack pointer ≡ A7

A7 is *not* used as a regular address register
A7 is used as a *system* stack pointer

If you are in supervisor mode,
  A7 = SSP (supervisor stack pointer)
  use USP to access the user stack pointer

If you are in user mode,
  A7 = USP (user stack pointer)
  SSP not visible

USP and SSP are physically different registers.
Subroutines

**JSR/BSR label**

Operation:

1. save PC (program counter) on the stack
   - i.e. save the address of the next instruction
   - why?

2. execution continues at specified address

In register transfer language (rtl):

(SP) ← (SP) − 4
((SP)) ← (PC)
(PC) ← address
Subroutines

Returning from subroutine:
1. Return from subroutine
   RTS
   → get PC from stack & return
   → used most of the time

2. Return & restore condition codes
   RTR
   → get condition codes from stack
   → get PC from stack & return

How did the condition codes get on the stack?
→ programmer put them on stack in the subroutine

in rtl:
(PC) ← ((SP))
(SP) ← (SP) + 4

in rtl:
(CCR) ← ((SP))
(SP) ← (SP) + 2
(PC) ← ((SP))
(SP) ← (SP) + 4
Subroutines

- stack grows _______ in memory
- stack pointer points to

---

Memory Map:
0000
03FF
0400

5000

---

ORG $400

start
CLR.B D0

... JSR max
MOVE.W D0,D1
... STOP #$2700

max ...
... RTS

data area

END start
Defining the system stack

-> assumes no operating system

- Memory map #1
  0000
  ...
  03FF
  0400

- Memory map #2
  0000
  ...
  03FF
  0400

????
Defining the system stack

1. If there is an operating system, the operating system will initialize the stack pointer.

2. If there is a monitor program, check the SSP/USP registers.
   - for EASy68K, SSP = SS = $01000000
     USP = US = $00FF0000
   - for Teesside, SSP = SS = $00A00000

3. If there is no monitor program or if the monitor program does not initialize the SSP/USP, it is the programmer’s responsibility to initialize the SSP/USP in the 1st line of the program.
   - if stack is at $10000, use LEA $10000,A7
     or LEA $10000,SP
Subroutines

- programmer’s responsibility to save registers when entering the S/R and to restore registers when exiting the S/R
  - save registers that ____________________________
  - do not save registers that ______________________

- save/restore registers with move multiple
  - MOVEM.L  register list,-(SP)  store order A7 -> A0;
    e.g.   MOVEM.L  D0-D3/A2-A5,-(SP)  
    D7 -> D0

  - MOVEM.L  (SP)+,register list  restore as D0 -> D7;
    e.g.   MOVEM.L  (SP)+,D0-D3/A2-A5  
    A0 -> A7
Subroutines ...
saving context

e.g.  ORG  $400
main  LEA  $10000,A7 ;initialize SP  ← not
        ;necessary for Easy68K
          ...
        JSR  find_min
        MOVE.L D0,D1
          ...
        STOP  #$2700

find_min ;save registers
          ...
          ...
          ...
        RTS
          ...
        ... data here ...
END     main

          ;restore registers
Reading, Expectations:

Reading:
- **M68000 Assembly Language** [pdf, 92p; N. Znotinas]
  - review operation of JSR, BSR, RTS
  - review operation of MOVEM
- Run **Stack_mem1.x68** and **Stack_mem2.x68** under the Easy68K simulator. To demonstrate the stack behaviour, I have loaded values into some of the registers. The mem1/mem2 examples use different memory mapping techniques. Single step through the programs. Make sure you understand how the stack is working. Use the **view stack** feature so you can see the stack and the program simultaneously. Can you see the stack and all of its entries? Can you see the program? For mem2, use the **view memory** feature to verify that both the program and the stack are located at $8000.

Expectations:
- you can use JSR, BSR, RTS, MOVEM correctly
- you can write subroutines correctly
- you can explain the stack structure for a program with multiple subroutines using diagrams to illustrate stack contents