Assembly Language and C

- We are interested in:
  - How a high-level language uses low-level language features?
  - C: System programming, device drivers, ...
  - Use of addressing modes by compilers
  - Parameter passing in assembly language
  - Local storage

Assembly Language and C, ACIA example

Character_Input(Func, Dev_loc, Input_Char, Error_St)
Error_St=0
IF Func = 0
THEN Initialize Input_Dev
ELSE Read status of Input_Dev
   IF status OK THEN
      BEGIN
         Set Cycle_Count to max value
         REPEAT
            Read status of Input_Dev
            Decrement Cycle_Count
            UNTIL Input_Dev is ready OR Cycle_Count = 0
         END
         IF Cycle_Count = 0
         THEN Error_St = $FF
         END_IF
      END
   ELSE Read status of Input_Dev
   END_IF
END_IF
End Character_Input

ACIA example, 68000 assembly language version

* ACIA_Initialize and Character_Input routine
* Data register D0 contains Function (zero=initialize, non-zero = get a character)
* Data register D2 is re-used for the Cycle_Count (a timeout mechanism)
* Data register D1 returns Error_Status
* Data register D0 returns the character from the ACIA
* Data register D3 is temporary storage for the ACIA's status
* Data register D4 is temporary storage for the masked ACIA's status (error bits)
* Address register A0 contains the address of the ACIA's control/status register

Char_In MOVEM.W D3-D4,-(A7) ; Push working registers on the stack
CLR.B D1 ; Start with Error_Status clear
CNST.H $0,0 ; IF Function not Zero THEN get input
BNE Input ; ELSE initialize ACIA
MOVUL.B $1,(A0) ; Reset the ACIA
MOVUL.B $19,(A0) ; Configure the ACIA
BRA Exit_2 ; Return after initialization
ACIA example, 68000 assembly language version

```
InPut  MOVE.W #$FFFF,D0    Set up Cycle_Count for time-out  
InPut1 MOVE.B  (A0),D3       Read the ACIA's status register
          MOVE.B  D3,D4         Copy status to D4
          AND.B   #%01111100,D4 Mask status bits to error conditions
          BNE     Exit_1        IF status indicates error, set error
          flags & return
          BTST   #0,D3         Test data_ready bit of status
          BNE     Data_Ok       IF data_ready THEN get data
          SUBQ.W  #1,D0                       ELSE decrement Cycle_Count
          BNE     InPut1        IF not timed out THEN repeat
          MOVE.B  #$FF,D1                        ELSE Set error flag
          BRA     Exit_2                              and return

Data_Ok  MOVE.B  (2,A0),D2     Read the data from the ACIA
          BRA     Exit_2        and return

Exit_1   MOVE.B  D4,D1         Return Error_Status
Exit_2   MOVEM.W (A7)+,D3-D4   Restore working registers
          RTS                   Return
```

Mechanisms for Parameter Passing

- **Passing parameters by value**
  - Actual parameter is transferred
  - If the parameter is modified by the subroutine, the "new value" does not affect the "old value"

- **Passing parameters by reference**
  - The address of the parameter is passed
  - There is only one copy of parameter
  - If parameter is modified, it is modified globally

Passing Parameters via Registers

- Two registers are used in subroutine and have to be saved on the stack:
  - `MOVE.W D3-D4,-(A7)` (otherwise, data would be lost)
- D0 is simply reused without saving, because the old data will not be needed

**PROS:**
- Position independent code
- Re-entrancy (subroutine has to save registers before they are reused)

**CONS:**
- Reduces number of registers available to programmer
- Number of parameters limited by the number of registers

Passing Parameters by Value

```
LEA     (-4,A7),A7 Save space on stack for Error_Status and Input_Char
          SP             0
          Error_Status  -2
          Input_Char    -4
          State of stack after executing this instruction

Address with respect to the initial stack pointer
```

```
BSR     Char_In Call subroutine
          LEA     (+4,A7),A7 Clean up stack - remove parameters
          Function:ACIA
          MOVE.W  (A7)+,Char Pull the input character off the stack
          MOVE.W  (A7)+,Err Pull the Error_Status off the stack
          SP             -14
          Function      -10
          Error_Status  -2
```

```
MOVE.L  #ACIA,-(A7) Push ACIA address on the stack
          SP             -10
          Input_Char    -8
          Error_Status  -2
          ACIA address  -4
          Push function code on the stack
```

```
MOVE.W  Func,-(A7) Push function code on the stack
          SP             0
          Function      10
          Error_Status  2
          Input_Char    4
          ACIA address  8
```
Passing Parameters by Value

* Character_Input and ACIA_Initialize routine
* Data register D3 is temporary storage for the ACIA's status
* Data register D4 is temporary storage for the ACIA's status
* Address register A0 contains the address of the ACIA's control/status register

 Character_Input: MOVEM.L A0/D3-D4,-(A7) Push working registers on the stack
 MOVE.L (18,A7),A0 Read address of ACIA from the stack
 CLR.B (24,A7) Start with Error_Status clear

Passing Parameters by Value

CMFL.E $7,(A7) IF Function not zero THEN get input
BRE Input
MOVEL.E $7,(A0)
MOVEL.E $9,(A0)
BRA Exit_2

Input: MOVE.W #3,(A0) Set up Cycle_Count for time-out
MOVE.W #$19,(A0)
BRA Exit_2

Passing Parameters by Value

Data_OK: MOVE.W (2,A0),(22,A7) Read the data from the ACIA and put on the stack
BRA Exit_2

Passing Parameters by Reference

PEA Func Push Function address on the stack
PEA ACIA Push ACIA address on the stack
PEA Error_Status Push address of Error_Status
PEA Char Push address of Input data
PEA Char_In Call subroutine
LEA #18(A7),A7 Clean up the stack - remove the four addresses

Passing Parameters by Reference

BSR Char_In Call subroutine
LEA #18(A7),A7 Clean up the stack - remove parameters

Passing Parameters by Reference

BSR Char_In Call subroutine
LEA #18(A7),A7 Clean up the stack - remove the 4 addr

Address with respect to the final value of stack pointer
Address with respect to the initial stack pointer

Passing Parameters by Reference

MOVE.L 32,(A7),A0  Read address of Char from the stack
MOVE.L 36,(A7),A1  Read address of Error_Status from the stack
MOVE.L 40,(A7),A2  Read address of ACIA from the stack
MOVE.L 44,(A7),A3  Read address of Function from the stack
CLR.B (A1)  Start with Error_Status clear
CMPI.B #0,(A3)  IF Function not zero THEN get input
BNE InPut  ELSE initialize ACIA
MOVE.B #3,(A2)
MOVE.B #$19,(A2)
BRA Exit_2  Return after initialization

* InPut MOVE.W #$FFFF,D0  Set up Cycle_Count for timeout
InPut1 MOVE.B (A2),D3  Read the ACIA's status register
MOVE.B D3,D4  Copy status to D4
AND.B #%01111100,D4  Mask status bits to error conditions
BNE Exit_1  IF error, set flags and return
BTST #0,D3  Test data_ready bit of status
BNE Data_OK  IF data_ready THEN get data
SUBQ.W #1,D0  ELSE decrement Cycle_Count
BNE InPut1  IF not timed out THEN repeat
MOVE.B #$FF,(A1)  ELSE Set error flag and return
BRA Exit_2

Saved registers
- SP
- Return address
- Char
- Error_Status
- ACIA address
- Function

-48
-20
-16
-12
-8
-4
0

The Stack and Local Variables

Subroutines often need local workspace.

- We can use a fixed block of memory space – static allocation – but:
  - The code will not be relocatable
  - The code will not be reentrant
  - The code will not be able to be called recursively

- Better solution: dynamic allocation
  - Allocate all local variables on the stack
  - STACK FRAME = a block of memory allocated by a subroutine to be used for local variables
  - FRAME POINTER = an address register used to point to the stack frame

The Stack and Local Variables

- LINK and UNLK automate the creation and removal of the stack frame
- Implementation
  - LINK: STP A1,16 - STP A1,8 - STP A1,4 - STP A1,2 - STP A1,0 - Move the content of the stack pointer to the stack
  - UNLK: LDP A1,16 - LDP A1,8 - LDP A1,4 - LDP A1,2 - LDP A1,0 - Restore the stack pointer

- It can be done simply by modifying the stack pointer:
  - AnySub: LEA (4,(A7),A0)  Set up address frame pointer
  - Return from subroutine
  - The stack frame
  - Collapses the stack frame and returns from subroutine
The Stack and Local Variables

Nested subroutines: A calls B, then B calls A

PEA Char
Push address of dest. for the input
PEA Error_Status
Push address of Error_Status message
PEA ACIA
Push ACIA's address on the stack
MOVE.W Function,-(A7)
Push value of function code on the stack
LEA (14,A7),A7
Call subroutine

* Character Input and ACIA_Initialize routine
* SF location A6 - 6 holds the ACIA's status
* SF location A6 - 4 holds the ACIA's masked status (error bits only)
* SF location A6 - 2 holds the Cycle_Count
* A1 contains the address of the Error_Status
* A2 contains the address of the ACIA's control/status register

Char_In
LINK A6,#-6
Create a stack frame for three words
MOVEM.L A1-A2,-(A7)
Push working registers on the stack
MOVEA.L (14,A6),A1
Read address of Error_Status from the stack
MOVEA.L (10,A6),A2
Read address of ACIA from the stack
CLR.B (A1)
Clear Error_Status
MOVE.W #$FFFF,(-2,A6)
Set up Cycle_Count for timeout
CMPI.B #0,(8,A6)
IF Function not zero THEN get input
BNE InPut
ELSE initialize ACIA
MOVE.B #3,(A2)
Reset ACIA
MOVE.B #$19,(A2)
Configure ACIA
BRA Exit_2
Return after initialization

InPut
MOVE.B (A2),(-4,A6)
Read the ACIA's status register — save in Temp1
MOVE.L (18,A6),(A1)
Get address for data dest. (reuse A1)
MOVE.B (2,A2),(A1)
Read data from ACIA
BRA Exit_2

Exit_1
MOVE.B (-6,A6),(A1)
Return Error_Status

Exit_2
MOVEM.L (A7)+,A1-A2
Restore working registers

C and The 68000

- Compiler and 68000 instruction set
- C data types and implementation
- Storage classes
- Functions and parameters
- Pointers
Compiling a C Program

```c
void main (void) {
  int i;
  int j;
  i = 1;
  j = 2;
  i = i + j;
}
```

* Comments

The 68000 family supports three basic data types:
- Bytes, Word, Longword
- Each can be interpreted as signed or unsigned

C built-in types:
- Integer, character, floating point, double-precision
- Void – refers to the null data type
- Implementation dependant!

<table>
<thead>
<tr>
<th>Data Type</th>
<th>C Name</th>
<th>Width (b)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>int</td>
<td>16</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>short integer</td>
<td>short int</td>
<td>8</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>long integer</td>
<td>long int</td>
<td>32</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned integer</td>
<td>unsigned int</td>
<td>16</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>character</td>
<td>char</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>single-precision floating point</td>
<td>float</td>
<td>32</td>
<td>10^-38 to 10^38</td>
</tr>
<tr>
<td>double-precision floating point</td>
<td>double</td>
<td>64</td>
<td>10^-300 to 10^300</td>
</tr>
</tbody>
</table>

C Data Types, cont'd

Local variables:
- Defined inside a function
- Cannot be accessed from outside the function
- Normally lost when a return from the function is made

Global variables:
- Defined outside a function
- Can be accessed both from inside and outside the function
- Variables defined in a block exist only within that block

```c
int i; /*global variable, visible to everything from this point*/
void function_1(void) /*A function with no parameters*/
{
  int k; /*Integer k is local to function_1*/
  {
    int q; /*Integer q exists only in this block*/
    int j; /*Integer j is local and not the same as j in main*/
  }
  void main(void) {
    int j; /*Integer j is local to this block within function main*/
  } /*This is the point at which integer j ceases to exist*/
```

Storage Class

Storage class specifiers:
- auto
  - Variable is no longer required once a block has been left; Default
- register
  - Ask compiler to allocate the variable to a register
- static
  - Allows local variable to retain its value when a block is reentered
    - Initialized only once, by the compiler!
- extern
  - Indicates that the variable is defined outside the block
  - The same global variable can be defined in more than one modul

```c
volatile

To define variables that can be changed externally
- Compiler will not put them in registers
- Think about Status Registers !
const
- Variable may not be changed during the execution of a program
- Cannot be changed unintentionally,
  but CAN be changed externally
  (as a result of an I/O, or OS operations external to the C program)
```

Type conversion

In C, done either automatically or explicitly (casting)

### Access Modifiers

- **volatile**
  - To define variables that can be changed externally
- **const**
  - Variable may not be changed during the execution of a program
- **extern**
  - Indicates that the variable is defined outside the block
  - The same global variable can be defined in more than one modul

Returning a Value from a Function

Example: main calls function adder
- adder function has 2 formal parameters (x and y)
- Formal parameters behave like local variables within the function
- When the function is called, formal parameters are replaced
  by the values of the actual parameters (a and b)

```c
# define adder(int a, int y); /* returns an integer */
void main (void) {
  int a, b, c; /* assign variables a, b, and c to regs */
  a = 1; b = 2; /* provide some dummy values for a and b */
  c = adder(a, b); /* c is assigned the integer returned by adder */
```
```c
*1 int adder(int x, int y)
  * Parameter x is at 8(A6)
  * Parameter y is at 10(A6)
  LINK A6,#0
  *2 return x + y;
  MOVE 8(A6),D1
  ADD 10(A6),D1
  MOVE D1,D0
  *3 }
  UNLK A6
  RTS

void main(void)
  * Parameter x is at 8(A6)
  * Parameter y is at 10(A6)
  * Parameter c is at 6(A6)
  _main
  LINK A6,#0
  *8 x = *y;
  MOV.A #0,-0(A6)
  LEA A0,-4(A6)
  ADDQ #2,A0
  MOVE 2(A0),-2(A6)
  UNLK A6
  RTS

int x = 2, y = 3;
swap(&x, &y); /* let’s swap a and b */
```

No, because this program is using a call-by-value mechanism.
Functions and Parameters, cont’d

*1 void swap (int a, int b)
* Parameter a is at 8(A6)
* Parameter b is at 10(A6)
* Variable temp is at −2(A6)
swap
LINK A6, #−2
*2 {
*3 int temp;
*4 temp = a;
MOVE 8(A6), #−2(A6)
*5 a = b;
MOVE 10(A6), #8(A6)
*6 b = temp;
MOVE #−2(A6), #10(A6)
} UNLK A6
RTS

*8 void main (void)
* Variable x is at −2(A6)
* Variable y is at −4(A6)
_main
LINK A6, #−4
*9 {
*10 int x = 2, y = 3;
MOVE #2, #−2(A6)
MOVE #3, #−4(A6)
*11 swap (x, y);
MOVE #3, #−(A7)
MOVE #2, #−(A7)
JSR swap
*12 }
UNLK A6
RTS

Functions and Parameters
USE OF STACK – call-by-value

To permit the function to modify the parameters, pass the address of the parameters
This will work...

/* swap two parameters in the calling program */
void swap (int *a, int *b) {
    int temp;
    *a = *b;
    *b = temp;
}

void main (void) {
    int x = 2, y = 3;
    swap (&x, &y);
}

Functions and Parameters
USE OF STACK – call-by-value, cont’d

Figure 3.11

Functions and Parameters
Call-by-reference

*1 void swap (int *a, int *b)
* Parameter a is at 8(A6)
* Parameter b is at 12(A6)
* Variable temp is at −2(A6)
swap
LINK A6, #−2
*2 {
*3 int temp;
*4 temp = *a;
MOVEA.L 8(A6), A4
MOVE (A4), #−2(A6)
*5 *a = *b;
MOVEA.L 12(A6), A0
MOVE (A0), (A4)
*6 *b = temp;
MOVEA.L 12(A6), A4
MOVE #−2(A6), (A4)
} UNLK A6
RTS

*8 main ()
* Variable x is at −2(A6)
* Variable y is at −4(A6)
_main
LINK A6, #−4
*9 {
*10 int x = 2, y = 3;
MOVE #2, #−2(A6)
MOVE #3, #−4(A6)
*11 swap (x, y);
MOVEA.L #−4(A6), (A6)
MOVEA.L #−2(A6), (A6)
JSR swap
*12 }
UNLK A6
RTS

Functions and Parameters
Call-by-reference, cont’d

Figure 3.12

Functions and Parameters
USE OF STACK, Call-by-reference

Figure 3.11

Functions and Parameters
Call-by-reference

To permit the function to modify the parameters, pass the address of the parameters
This will work...

/* swap two parameters in the calling program */
void swap (int *a, int *b) {
    int temp;
    *a = *b;
    *b = temp;
}

void main (void) {
    int x = 2, y = 3;
    swap (&x, &y);
}