Student: “Programming in assembly is a lot like digging a post hole with a teaspoon. Of course, you do have complete control over every piece of dirt”

Valvano: “Programming in a high level language is a lot like painting a masterpiece with a paint roller. Of course, you do get the job done quickly.”

8. Program structures

Chapter 8 objectives are to:
- Explain how to define local variables on the stack,
- C compilers local variables and pass parameters,
- call by value versus call by reference parameter passing,
- Use linked structures to implement finite state machines,
- Implement system calls using software interrupts,

8.1. Local Variables

8.1.1. Introduction

Scope => from where can this information be accessed
- local means restricted to current program segment
- global means any software can access it

Allocation => when is it created, when is it destroyed
- dynamic allocation using registers or stack
- permanent allocation assigned a block of memory

A local variable (local scope, dynamic allocation)
- temporary information
- used only by one software module
- allocated, used, then deallocated
- not permanent
- implement using the stack or registers

Reasons why we place local variables on the stack include
- dynamic allocation/release allows for reuse of memory
- limited scope of access provides for data protection
- only the program that created the local can access it
- the code is reentrant
- the code is relocatable
- the number of variables is more than registers

****************************************************************************

short A1; int A2;
a) What is the difference between $A1$ and $A2$?

$\text{short}$ is always 16 bits,
while $\text{int}$ is compiler-dependent.
compilers allow you to specify the size of $\text{int}$.
The proper use of $\text{int}$ is for situations
where the size doesn’t matter and
you wish compiler to choose most efficient size
Compiler chooses size based on the microcomputer
MC6805 uses 8-bit,
6812 uses 16-bit, and
Pentium uses 32-bit).
Good uses for $\text{int}$
TRUE (any nonzero) and FALSE (zero).
\[
\text{for}(I=0; I<10; I++\{}\}
\]

*************************************************
$\text{short} \ B1; \ \text{short static} \ B2$;
b) What is the difference between $B1$ and $B2$?

$B1$ is public, and can be accessed from anywhere
any file can access $B1$ using
\[
\text{extern \ short} \ B1;
\]
$B2$ is private,
can be only be accessed from software in this file.

*************************************************
$\text{short} \ C1; \ \text{short volatile} \ C2$;
c) What is the difference between $C1$ and $C2$?

$C2$ will not be optimized by the compiler
$C1$ can be optimized
it will not keep a copy of $C2$ in a register,
reload a new value each time it is needed.
$C2$ can be changed by operations
outside direct software action of this program.
Two good applications of volatile are
I/O ports
global variables shared by two or more threads.
If $\text{SCISR1}$ were not volatile then
\[
\text{unsigned \ char} \ \text{InChar}(\text{void})\{
\begin{align*}
\text{while} \ ((\text{SCISR1} \ & \text{RDRF}) \ == \ 0)\{}; \\
\text{return} \ (\text{SCIDRL})\{};
\end{align*}
\]
could be "optimized" to
\[
\text{InChar::} \ \text{lda} \ SCISR1 \quad \text{copy of SC0SR1}
\text{loop::} \ \text{bita} \ #80 \quad \text{check for RDRF}
\text{beq} \ \text{loop}
\text{ldab} \ SCIDRL
\text{clr}
\]

Jonathan W. Valvano
rts

short D1=5; short const D2=5;
d) What is the difference between D1 and D2?
   Formally, const means can't be changed.
   On an embedded system,
   it means D2 is stored in ROM, and
   D1 is stored in RAM.
   In C, there will be two copies of D1.
   The initial value is stored in ROM and
   the actual working copy of D1 is stored in RAM.

void program(void){
    short e1; short static e2;
}
e) What is the difference between e1 and e2? When used inside a function static has a
different meaning than when used outside a function. In this situation,
   e2 is statically allocated in permanent RAM and
   the values persist from one function call to the next.
   e1 is dynamically
       allocated on the stack,
       used inside the function, and
       deallocated at the end of the function.
   Values of e1 do not persist from one function call to next.

short F1(short n){ return n+1;}
short static f2(short n){ return n+1;}
f) What is the difference between F1 and f2? In this context, static is used in a similar
manner to the way static is used for a variable outside a function.
   F1 is public,
       can be called from anywhere in the software system.
   extern short F1(short n);
   f2 is private, and
       can be only be called from software in this file.

char FreezingPoint1=32;
char FreezingPoint2=0x20;  // degrees F
FreezingPoint1 fcb 32
FreezingPoint2 fcb 0x20
g) What is the difference between FreezingPoint1 and FreezingPoint2?
   The only difference is style.
   32 is much better style (easy to read) than 0x20.
Similarly
\begin{align*}
\text{\texttt{bset PTT, #0x20}} & \quad \text{is easy to understand} \\
\text{\texttt{bset PTT, #32}} & \quad \text{is harder to understand}
\end{align*}

![Empty Stack vs Stack with 3 elements](image)

\textit{Figure 8.1. 6811 and 6812 stack.}

The \texttt{tsx} and \texttt{tsy} instructions do not modify the stack pointer.

![Stack before vs Stack after \texttt{tsx}](image)

\textit{Figure 8.2. The \texttt{tsx} instruction creates a stack frame pointer.}

The LIFO stack has a few rules:
1. Program segments should have an equal number of pushes and pulls;
2. Stack accesses (PUSH or PULL) should not be performed outside the allocated area;
3. Stack reads and writes should not be performed within the free area,
   PUSH should first decrement SP, then store the data,
   PULL should first read the data, then increment SP.

\subsection{8.1.2. Implementation}
Stack implementation of local variables has four stages:
\begin{itemize}
  \item binding
  \item allocation
\end{itemize}
1. **Binding** is the assignment of the address (not value) to a symbolic name.

   \texttt{sum set 0 \ 16-bit local variable}

2. **Allocation** is the generation of memory storage for the local variable.

   \texttt{pshx allocate sum}

   In this next example, the software allocates the local variable by decrementing the stack pointer. This local variable is also uninitialized.

   \texttt{des allocate sum des}

   If you wished to allocate the 16-bit local and initialize it to zero, you could execute.

   \texttt{movw \#0,2,-sp}

   This example allocates 20 bytes for the structure \texttt{big[20]}.

   \texttt{leas -20,sp allocate big[20]}

3. The **access** to a local variable is a read or write operation that occurs during execution. In the next code fragment, the local variable \texttt{sum} is set to 0.

   \texttt{movw \#0,sum,sp}

   In the next code fragment, the local variable \texttt{sum} is incremented.

   \texttt{ldd sum,sp addd \#1 std sum,sp sum=sum+1}

4. **Deallocation** is the release of memory storage for the location variable.

   \texttt{pulx deallocate sum}

   In this next example, the software deallocates the local variable by incrementing the stack pointer.

   \texttt{ins \ins deallocate sum}

   In this last example, the technique provides a mechanism for allocating large amounts of stack space.
leas 20,sp deallocate big[20]

Example of local variables on stack
org $4000
* calculate sum of numbers
* Input: RegD num
* Output: RegD Sum of 1,2,3,...,num
* Errors: may overflow
* 1) binding
num set 2  loop counter 1,2,3
sum set 0  running
calc
* 2) allocation
  pshd           allocate num
  movw #0,2,-sp  sum=0

* 3) access
loop ldd sum,sp
  addd num,sp
  std sum,sp  sum = sum+num
  ldd num,sp
  subd #1
  std num,sp  num = num-1
  bne loop
  ldd sum,sp  result

*4) deallocate
  leas 4,sp
  rts

main lds #$4000
  ldd #100
  jsr calc
  bra *
  org $FFFFE
  fdb main

Draw a stack picture
1) in text form
   SP  ->  sum
   SP+2   ->  num
   SP+4   ->  return address

2) graphically
3) using TExaS

*******Run local.rtf on TExaS**********