Recap
Finite State Machines
Pointer implementation

Overview
Fixed-point: why, when, how
Local variables: scope and allocation
How these concepts apply to C
Binding, allocation, access, deallocation

Floating point numbers
ANSI/IEEE Std 754-1985
single-precision (32-bit),
double-precision (64-bit), and
double-extended precision (80-bits).

The floating-point format
Bit 31        sign, s=0 for positive, s=1 for negative
Bits 30:23    8-bit biased binary exponent 0 ≤ e ≤ 255
Bits 22:0     24-bit mantissa, m
expressed as a binary fraction
a binary 1 as the most significant bit is implied
m = 1.m1m2m3...m23

\[ f = (-1)^s \times 2^{e-127} \times m \]

10.1. Fixed-point numbers
Fixed point numbers

Why? (wish to represent non-integer values)
Next lab measures distance from 0 to 3 cm
E.g., 1.234 cm

When? (range is known, range is small)
Range is 0 to 3 cm
Resolution is 0.003 cm

How? (value = Integer*Δ)
16-bit unsigned integer
Δ = 10^-7 decimal fixed-point
Range becomes 0.000 to 65.535

Output an integer.
Assume integer,
n, is between 0 and 9999.

\[ \text{not very pretty} \]
\[ \text{OutChar($30+n/1000)$} \] ;thousand’s digit
\[ n = n\%1000 \]
\[ \text{OutChar($30+n/100)$} \] ;hundred’s digit
\[ n = n\%100 \]
\[ \text{OutChar($30+n/10)$} \] ;ten’s digit
\[ n = n\%10 \]
\[ \text{OutChar($30+n$)} \] ;one’s digit

Output a fixed-point number.
Assume the integer part of the fixed point number,
n, is between 0 and 9999.

\[ \text{very pretty} \]

Jonathan W. Valvano
7.3. Local Variables

Introduction

**Introduction**

**scope** => from where can this information be accessed

- **private** means restricted to current program segment
- **public** 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 (private 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

**Registers are local variables**

**Allocation:** Register assigned to a task

**Access:** Register is used

**Deallocation:** Register free for other tasks

<table>
<thead>
<tr>
<th>Line</th>
<th>Program</th>
<th>RegB</th>
<th>RegX</th>
<th>RegY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Main lds #$4000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>bsr Timer_Init</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ldab #$FC</td>
<td>$FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>stab DDRT</td>
<td>$FC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ldx #goN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>FSM ldab OUT,x</td>
<td>Output</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>lslb</td>
<td>Output</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>lslb</td>
<td>Output</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>stab PTT</td>
<td>Output</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ldy WAIT,x</td>
<td>Pt</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>bsr Timer_Wait10ms</td>
<td>Pt</td>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ldab PTT</td>
<td>Input</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>andb #$03</td>
<td>Input</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>lslb</td>
<td>Input</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>abx</td>
<td>Input</td>
<td>Pt</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>ldx NEXT,x</td>
<td>Pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>bra FSM</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program 7.1. Register assignments in a finite state machine controller.

Jonathan W. Valvano
**Global variables in C**
- Defined outside of the functions
- Exist forever in RAM
- Public scope (accessed anywhere)
- Initialized at startup
  - Initialized to zero if not specified
  - Can define explicit initialization

```c
short Data;       // can be accessed by any program
                 // permanent allocation
                 // initialized to zero
short Count=10;  // can be accessed by any program
                 // permanent allocation
                 // initialized to ten at startup
```

**Local variables in C**
- Defined immediately after an open brace.
- Exist temporarily (in registers or on stack)
  - Created
  - Used
  - Destroyed
- Scope restricted to that program segment.
- Can be initialized each time segment begins
  - Not initialized if not specified
  - Can define explicit initialization

```c
void function(void){
  short i;         // scope restricted to function
  // temporary allocation
  // not initialized
  i = 10;
  while(i){
    short j=5;     // scope restricted to while loop
    // temporary allocation
    // initialized each time in loop
    i--;
  }
}
```

**Static variables in C**
- Defined in same places as globals or locals.
- Exist forever in RAM
- Scope restricted
  - To programs in that file
  - To program segment.
- Initialized at startup
  - Initialized to zero if not specified
  - Can define explicit initialization

```c
short static Mode;  // accessed only within this file
                    // permanent allocation
void function(void){
  short static Life=1000;  // initialized once
  // scope restricted to function
  // permanent allocation
  Life++;
  if(Life == 0) voidWarranty();
}
```
Stack usage

![Empty Stack and Stack with 3 Elements Diagram]

Figure 7.1. The 9S12 stack.

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

![Stack before and after \texttt{tsx} Diagram]

Figure 7.2. 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.

7.3. Local variables allocated on the stack

Stack implementation of local variables has four stages:
- binding
- allocation
- access, and
- deallocation.

1. \textbf{Binding} is the assignment of the address (not value) to a symbolic name.
   \texttt{sum set 0 ;16-bit local variable}

2. \textbf{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}
   \texttt{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 \textit{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}
\texttt{addd \#1}
\texttt{std sum,sp ;sum=sum+1}

4. \textbf{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}
\texttt{ins ;deallocation sum}

In this last example, the technique provides a mechanism for allocating large amounts of stack space.

\texttt{leas 20,sp ;deallocation big[20]}

\textbf{Example of local variables on stack}

\begin{verbatim}
short calc(short in){ short num,sum;
   sum = 0; num = in;
   while(num){
      sum = sum+num;
      num = num-1;
   }
   return sum;
}

go $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=in
   movw \#0,2,-sp ;sum=0

; 3) access
loop ldd sum,sp
   addd num,sp
\end{verbatim}
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 ; return result in Reg D

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 on TExaS**********

The bottom line
Scope specifies which module can access
   limiting scope reduces complexity
Allocation specifies where data is located
   Temporary register,
   Permanent RAM (data rmb 2)
   Temporary RAM (pt = malloc(100));
   Permanent ROM (list fcb 5, 6, 10, 9)
   Temporary on stack
Binding, allocation, access, deallocation