Overview

- Variable types
- Memory sections in C
- Parameter passing
- Stack frames

Types of Variables in C

1. Global variables: The variable that are declared outside a function
   - Exist during the execution of the program
2. Local variables: The variables that are declared in a function.
   - Exist during the execution of the function only
3. Static variables.
   - Can be either global or local.
   - A global static variable is valid only within the file where it is declared
   - A local static variable still exists after the function returns

Variable Types and Memory Sections

- Global variables occupy their memory space during the execution of the program
  - Need the static memory which exists during the program’s lifetime
- Static local variables still occupy their memory space after the function returns.
  - Also need the static memory which exists after the function returns.
- Local variables occupy their memory space only during the execution of the function.
  - Need the dynamic memory which exists only during the execution of the function
- So the entire memory space need be partitioned into different sections to be more efficiently utilized.
An Example

#include <stdio.h>
int x, y; /* Global variables */
static int b[10]; /* Static global array */
void auto_static(void)
{
    int autovar=1; /* Local variable */
    static int staticvar=1; /* Static local variable */
    printf(autovar = %i, staticvar = %i
, autovar, staticvar);
    ++autovar;
    ++staticvar;
}
void auto_static(void)
{
    int i; /* Local variable */
    for (i=0; i<5; i++)
    {
        auto_static();
    }

An Example (Cont.)

Program output:

Autovar = 1, staticvar = 1
Autovar = 1, staticvar = 2
Autovar = 1, staticvar = 3
Autovar = 1, staticvar = 4
Autovar = 1, staticvar = 5

Memory Sections in C for General Microprocessors

- Heap: Used for dynamic memory applications such as malloc() and calloc()
- Stack: Used to store return address, actual parameters, conflict registers and local variables and other information.
- Uninitialized data section .bss,
  - contains all uninitialized global or static local variables.
- Data section .data,
  - Contains all initialized global or static local variables
- Text section .text
  - Contains code
Memory Sections in WINAVR (C for AVR)

- Additional EEPROM section .eeprom
  - Contains constants in eeprom
- The text section .text in WINAVR includes two subsections .initN and .finiN
  - .initN contains the startup code which initializes the stack and copies the initialized data section .data from flash to SRAM.
  - .finiN is used to define the exit code executed after return from main() or a call to exit().

C Functions

```c
void main(void) {
    int i, j, k, m;
    i = mult(j, k);
    ...;
    m = mult(i, i);
    ...;
}

int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product = product + mcand;
        mlier = mlier - 1;
    }
    return product;
}
```

Two Parameter Passing Approaches

- Pass by value
  - Pass the value of an actual parameter to the callee
  - Not efficient for structures and array
    - Need to pass the value of each element in the structure or array
- Pass by reference
  - Pass the address of the actual parameter to the callee
  - Efficient for structures and array passing

Parameter Passing in C

- Pass by value for scalar variables such as char, int and float.
- Pass by reference for non-scalar variables i.e. array and structures.
C Functions (Cont.)

Questions:
• How to pass the actual parameters by value to a function?
• How to pass the actual parameters by reference to a function?
• Where to get the return value?
• How to allocate stack memory to local variables?
• How to deallocate stack memory after a function returns?
• How to handle register conflicts?

Rules are needed between caller and callee.

Parameter Passing and Return Value

• May use general registers to store part of actual parameters and push the rest of parameters on the stack.
  - WINAVR uses general registers up to r24 to store actual parameters
  - Actual parameters are eventually passed to the formal parameters stored on the stack.
• The return value need be stored in designated registers
  - WINAVR uses r25:r24 to store the return value.

Register Conflicts

• If a register is used in both caller and callee and the caller needs its old value after the return from the callee, then register conflict occurs.
• Compiler or assembly programmers need to check for register conflict.
• Need to save conflicts registers on the stack.
• Caller or callee or both can save conflict registers.
  - In WINAVR, callee saves conflict registers.

Stack Structure

• A stack consists of stack frames.
• A stack frame is created whenever a function is called.
• A stack frame is freed whenever the function returns.
• What’s inside a stack frame?
Stack Frame

A typical stack frame consists of the following components:

- Return address
  - Used when the function returns
- Conflict registers
  - Need to restore the old contents of these registers when the function returns
  - One conflict register is the stack frame pointer
- Parameters (arguments)
- Local variables

Implementation Considerations

- Local variables and parameters need be stored contiguously on the stack for easy accesses.
- In which order the local variables or parameters stored on the stack? In the order that they appear in the program from left to right? Or the reverse order?
  - C compiler uses the reverse order.
- Need a stack frame register to point to either the base (starting address) or the top of the stack frame
  - Points to the top of the stack frame if the stack grows downwards. Otherwise, points to the base of the stack frame (Why?)
  - WINAVR uses Y (r29: r28) as a stack frame register.

An Sample Stack Frame Structure for AVR

```c
int main(void) {
    ...
    foo(arg1, arg2, ..., argm);
}

void foo(arg1, arg2, ..., argm)
{
    int var1, var2, ..., varn;
    ...
}
```

A Template for Caller

Caller:

1. Store actual parameters in designated registers and the rest of registers on the stack.
2. Call the callee.
A Template for Callee

Callee:
1. Prologue
2. Function body
3. Epilogue

A Template for Callee (Cont.)

Prologue:
- Store conflict registers, including the stack frame register Y, on the stack by using push
- Pass the actual parameters to the formal parameters on the stack
- Update the stack frame register Y to point to the top of its stack frame

Function body:
Does the normal task of the function.

Epilogue:
1. Store the return value in designated registers r25:r24.
2. Deallocate local variables and parameters by updating the stack pointer SP.
   - SP=SP + the size of all parameters and local variables.
3. Restore conflict registers from the stack by using pop
   - The conflict registers must be popped in the reverse order that they are pushed on the stack.
   - The stack frame register of the caller is also restored.
   - Step 2 and Step 3 together deallocate the stack frame.
4. Return to the caller by using ret.

An Example

```c
int foo(char a, int b, int c);

int main()
{ int i, j;
i=0;
j=300;
foo(1, i, j);
return 0;
}

int foo(char a, int b, int c)
{ int x, y, z;
x=a+b;
y=c–a;
z=x+y;
return z;
}
```
### Stack frames for main() and foo()

<table>
<thead>
<tr>
<th>RAMEND</th>
<th>Return address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j</td>
</tr>
<tr>
<td></td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>Stack frame pointer Y for main()</td>
</tr>
<tr>
<td></td>
<td>r28</td>
</tr>
<tr>
<td></td>
<td>r29</td>
</tr>
<tr>
<td></td>
<td>r28</td>
</tr>
<tr>
<td></td>
<td>r29</td>
</tr>
<tr>
<td></td>
<td>z</td>
</tr>
<tr>
<td></td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td></td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>Empty</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conflict register Y (r29:r28)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack frame pointer Y for foo()</td>
</tr>
</tbody>
</table>

### An Example (Cont.)

**main:**

```assembly
ldi r28, low(RAMEND-4) ; 4 bytes to store local variables i and j
ldi r29, hi8(RAMEND-4) ; The size of each integer is 2 bytes
out SPH, r29          ; Adjust stack pointer so that it points to the new stack top.
out SPL, r28
clr r0
std Y+1, r0           ; The address of i in the stack is Y+1
std Y+2, r0
ldi r24, low(300)     ; The next four instructions implement j=300
ldi r25, high(300)    
std Y+3, r24           ; r21:r20 keep the actual parameter j
std Y+4, r25           ; r23:r22 keep the actual parameter i
ldd r20, Y+3
std Y+5, r24           ; r24 keeps the actual parameter 1
ldd r21, Y+4
ldd r22, Y+1
ldd r23, Y+2
ldi  r24,low(1)       ; Call foo
rcall foo              ; r25:r24 keep the actual parameter j
...```

**foo:**

```assembly```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```