TI-RSLK

Texas Instruments Robotics System Learning Kit

Texas Instruments

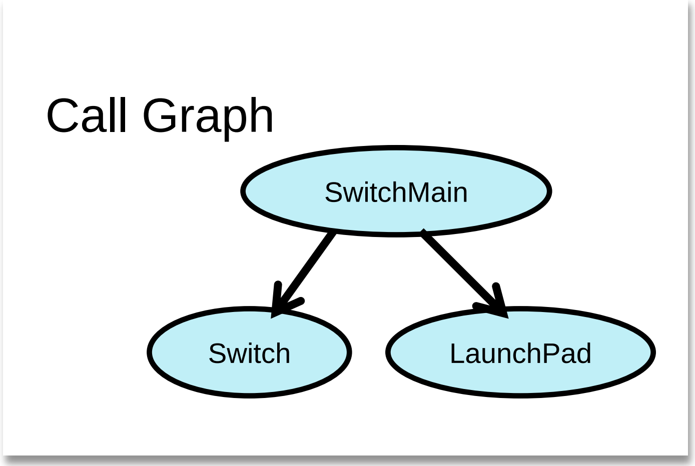
Module 4

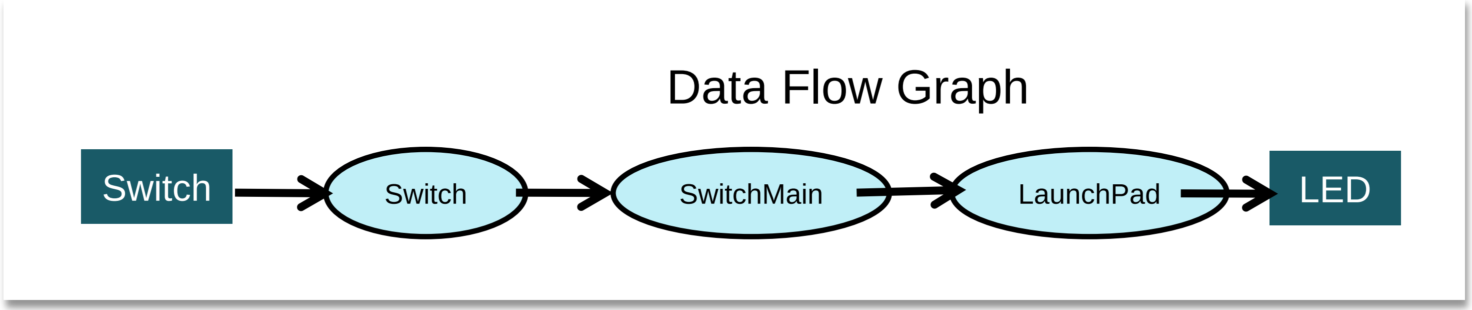
Lecture: Software Design using MSP432 - Design

Software Design using MSP432

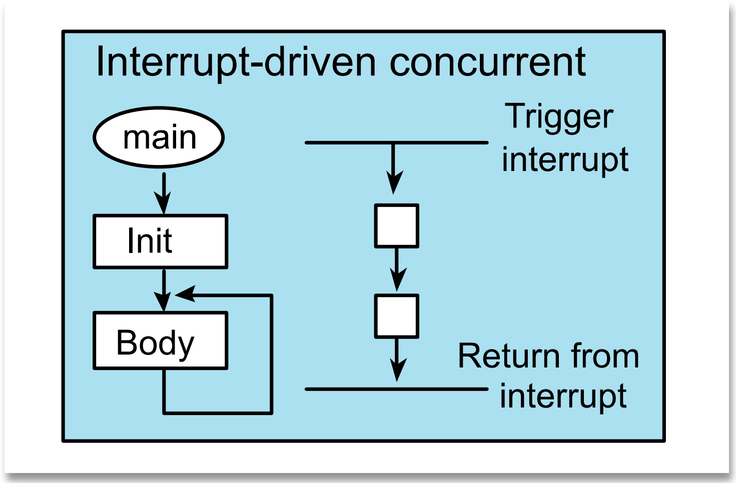
You will learn in this module

- Software Design
  - Call graph
  - Data Flow Graph
  - Successive refinement
  - Abstraction (functions)
  - Modular design (header/code files)





Multi-threading



System Design

What does being in a state mean?

- List state parameters

What is the starting state of the system?

- Define the initial state

What information do we need to collect?

- List the input data

What information do we need to generate?

- List the output data

How do we move from one state to another?

- Actions we could do

What is the desired ending state?

- Define the ultimate goal

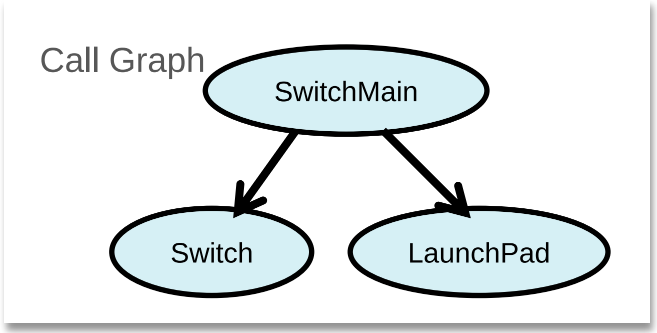
Successive Refinement

- Start with a task
  - Clear and unambiguous description: requirements, specifications
- Decompose the task into a set of simpler subtasks (components)
  - Subtasks are decomposed into even simpler sub-subtasks
  - Each subtask is simpler than the task itself
- Make design decisions
  - Document decisions and subtask requirements
- Ultimately, subtask is so simple, it can be implemented
  - Implementation
  - Testing
  - Documentation
- Combine components to build system
  - Interfaces are key

Three similar terms:
- Successive Refinement
- Stepwise Refinement
- Systematic Decomposition

Header files

- Why do we have header/code files?
  - Complexity abstraction
  - Separate what it does (header) from how it works
  - Automatic documentation (doxygen)
- What is in a header file?
  - Prototypes for public functions
  - Comments on what it does/how to use it
  - Code to make it load once
  - Shared structure
- What is not in a header file?
  - Function definitions
  - Variables
  - Anything private



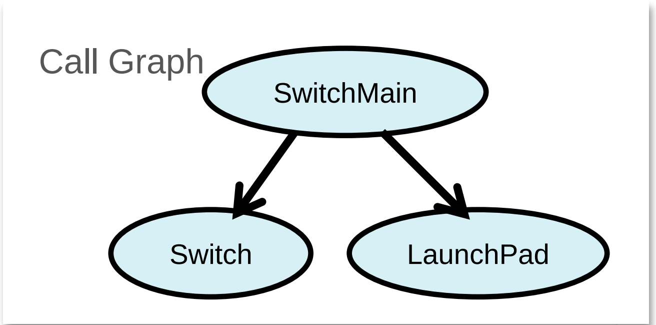
```
/**
 * @file      Switch.h
/**
 * Input from positive logic switch
 * interfaced to GPIO Port 1 bit 5.
 *
 * @param  none
 * @return 0x20 if pressed; 0x00 if not pressed
 * @brief  Switch input
 */
uint32_t Switch_Input(void);
```

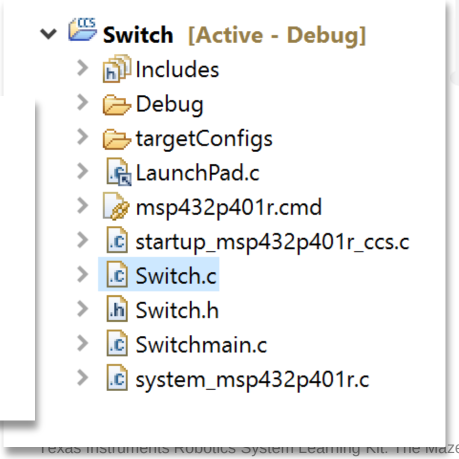
Code files

- What is in a code file?
  - Implementations for public functions
  - Variables
  - Private functions
  - Comments how it works
  - Comments on how it was tested
  - Comments on how it can be changed
- What is not in a code file?
  - References to private data/functions in other files

```
//------------Switch_Input-------
// Read and return P1.5
// Input: none
// Output: 0x20 if P1.5 is high
//         0x00 if P1.5 is low
uint32_t Switch_Input(void){
// read P1.5 input
  return (P1->IN&0x20);
// return 0x20(if pressed)
// or 0(if not pressed)
}
```

```
#include <stdint.h>
#include "Switch.h"
#include "../inc/LaunchPad.h"
```

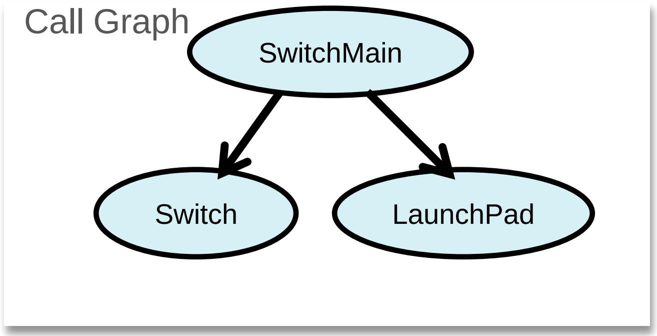


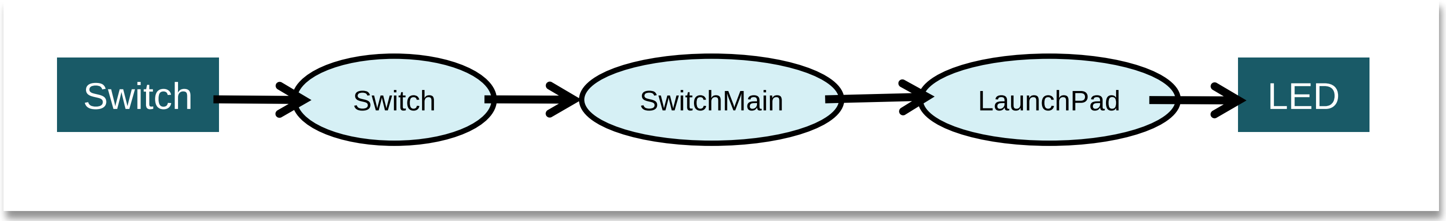


Software Design using MSP432

Summary

- Software design
  - Successive refinement
  - doxygen
  - Header/code files
  - Abstraction





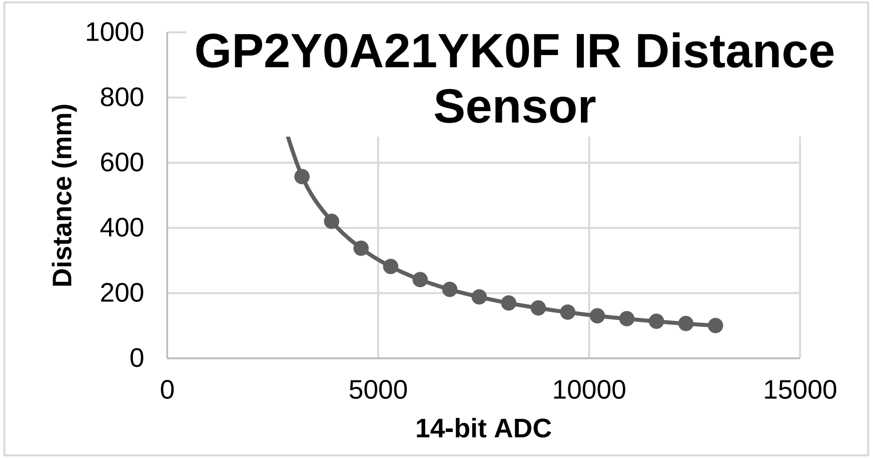
Module 4

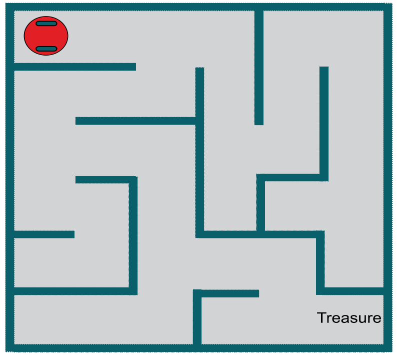
Lecture: Software Design using MSP432 - C Programming

C Programming on the MSP432

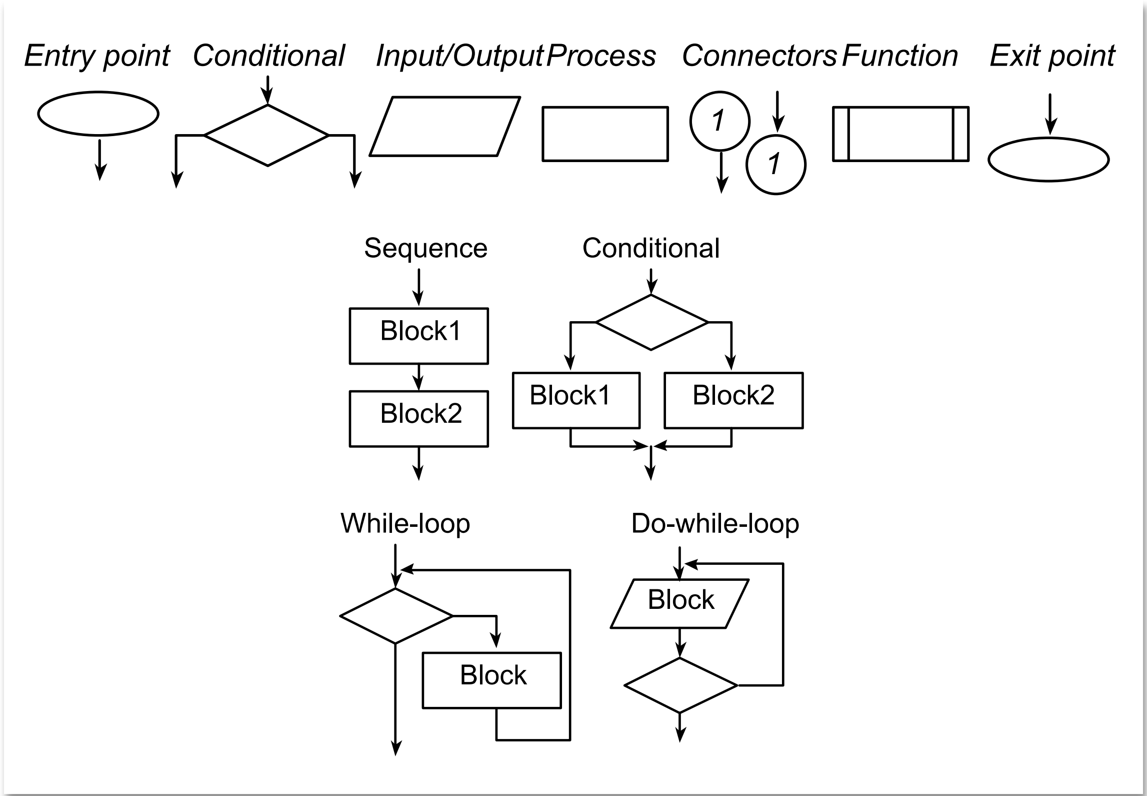
You will learn in this module

- Basics of C programming
  - Logic/shift operations
  - Arithmetic calculations
  - Conditionals
  - Loops
  - Functions
  - Variables
  - Constants
- Algorithm development (lab)
  - GP2Y0A21YK0F IR distance sensor
  - Where in the world am I?





Flowcharts


Entry point
Conditional
Input/Output
Process
Connectors
Function
Exit point
1
1
Sequence
Block1
Block2
Conditional
Block1
Block2
While-loop
Block
Do-while-loop
Block

Logic Operations

A	B	A&B	A|B	A^B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

A	~A
0	1
1	0

AND

- Select bits (AND with 1)
- Clear bits (AND with 0)

```
y = P1->IN&0x03; // select bits 1,0
x = x&(~0x08);   // clear bit 3
x &= ~0x08;      // clear bit 3
```

OR

- Combine
- Set bits (OR with 1)

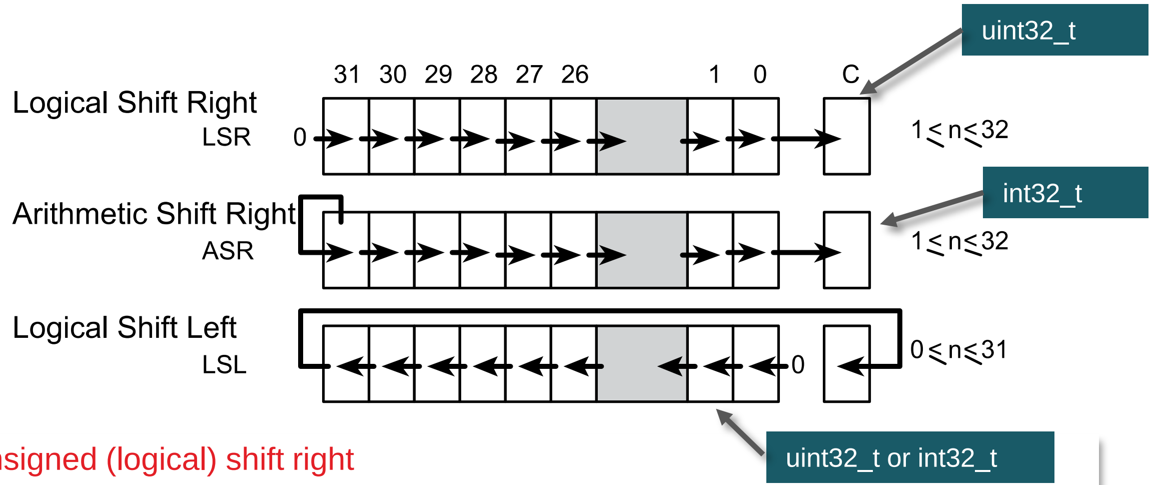
```
z = x|y;         // combine x,y
x = x|0x08;      // set bit 3
x |= 0x08;       // set bit 3
```

EOR

- Toggle bits (EOR with 1)

```
P1->OUT ^= 0x08;  // toggle bit 3
```

Shift Operations



Unsigned (logical) shift right

- Divide by 2^n
- Align bits

Signed (arithmetic) shift right

- Divide by 2^n

Shift left (logical/arithmetic)

- Multiply by 2^n
- Align bits

```
y = x>>3;       // divide by 8
```

```
x = P1->IN&0x01;     // P1.0 (0,1)
y = P2->IN&0x08;     // P2.3 (0,8)
z = (x<<1)|(y>>3);   // combine
```

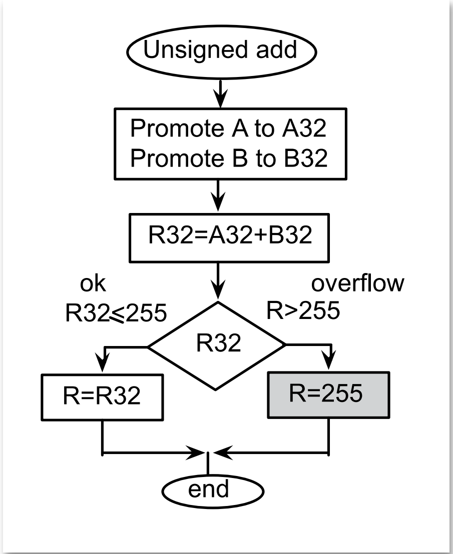
```
y = x<<8;  // multiply by 256
```

Arithmetic Operations

- Addition/subtraction
  - Two n-bit → n+1 bits
- Multiplication
  - Two n-bit → 2n bits
- Division
  - Avoid divide by 0
  - Watch for dropout
- Avoid overflow
  - Restrict input values
  - Promote to higher, perform, check, demote
- Signed versus unsigned
  - Either signed or unsigned, not both
  - Be careful about converting types

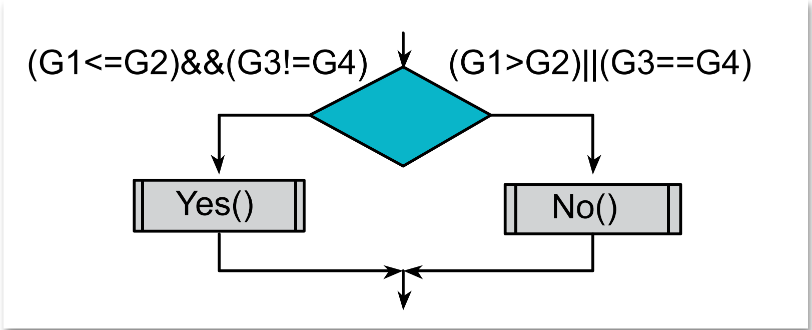
```
uint8_t Add(uint8_t A, uint8_t B){
uint32_t A32,B32,R32;
  A32 = A; B32=B; // promotion
  R32 = A+B;      // 32-bit addition
  if(R32>255){
    R32 = 255;    // ceiling
  }
  return R32;     // demotion
}
```

uint8_t	int8_t
uint16_t	int16_t
uint32_t	int32_t



Conditionals

- Boolean
  - Zero is false
  - Nonzero is true
  - && || ! are operators
- Relational
  - Compare similar types
  - Returns a Boolean
  - > >= < <= == !=
- Conditional
  - if-then
  - if-then-else



```
if((G1<=G2)&&(G3!=G4)){
  Yes();
}else{
  No();
}
```

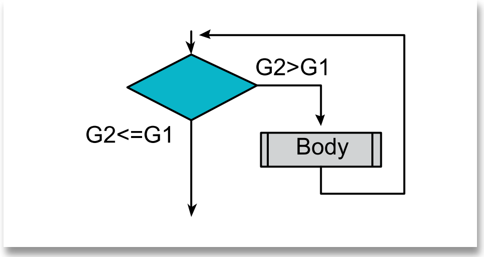
```
if(P1->IN&0x80){
  Something(); // if P1.7 is high
};
```

These are different `&` `&&`
These are different `|` `||`

while loops

while loop

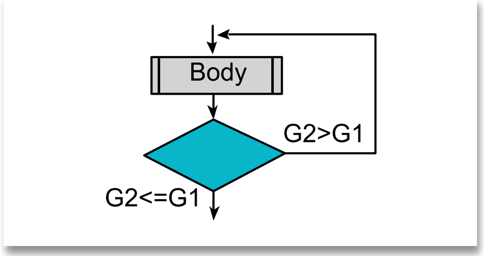
- Test first



```
while(G2>G1){
  Body();
}
```

do-while loop

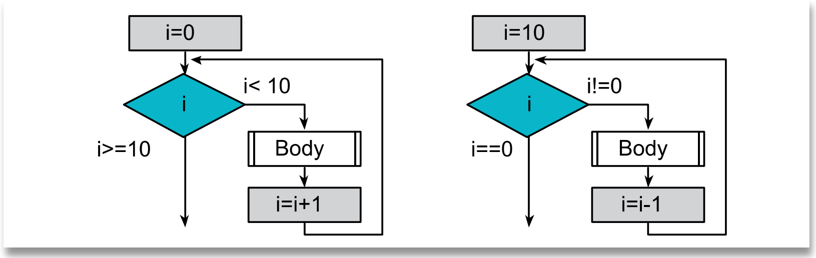
- Test last



```
do{
  Body();
} while(G2>G1);
```

for loop

- Test first



```
for(i=0; i<10;
i++){
  Body();
}
```

```
for(i=10; i!=0; i--){
  Body();
}
```

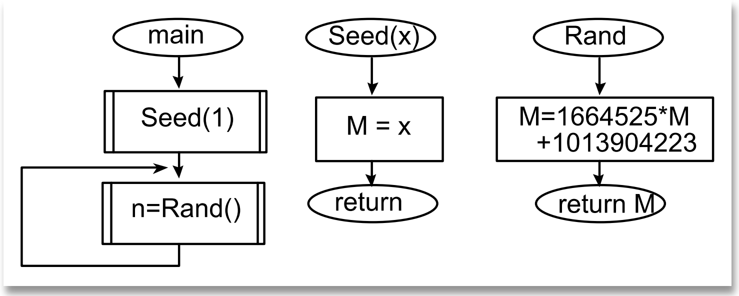
Functions

- What it does
  - Prototype
  - Header file

```
// random.h
void Seed(uint32_t x);
uint8_t Rand(void);
```

- How it works
  - Implementation
  - Code file

- Invocation
  - Calling sequence
  - Inputs: call by value/reference
  - Output: return value



```
// random.c
uint32_t static M;
void Seed(uint32_t x){
  M = x;
}
uint8_t Rand(void){
  M=1664525*M+1013904223;
  return M>>24;
}
```

```
// main.c
uint8_t n;
void main(void){
  Seed(1);
  while(1){
    n = Rand();
  }
}
```

Examples of variables

- Global
  - Public scope
  - Permanent allocation
  - Bad style
- Static
  - Private scope to file
  - Permanent allocation
  - Sharing: ISR ↔ Functions
- Local - Automatic
  - Private scope,
  - Dynamic allocation
- Static local
  - Private scope to function
  - Permanent allocation

```
uint32_t static M;
void Seed(uint32_t x){
  M = x;
}
uint8_t Rand(void){
uint32_t n;
uint32_t static count=0;
  count++;
  M=1664525*M+1013904223;
  n = M>>24;
  return (uint8_t)n;
}
```

```
uint8_t global;
void main(void){
uint8_t n;
  Seed(1);
  while(1){
    n = Rand();
  }
}
```

Variables

Scope => from where can it be accessed

- Private means restricted, need to know basis
  - More protection, simpler systems
- Public means any software can access it
  - Difficult to debug, hidden complexity

Allocation => when is it created & destroyed

- Dynamic allocation using registers or stack
- Permanent allocation assigned a block of memory

Type

- Signed/unsigned
- Precision: 8, 16, 32 bits

Can you convert between types?

uint8_t	→ uint16_t, int16_t, uint32_t, int32_t
int8_t	→ int16_t, int32_t
uint16_t	→ uint32_t, int32_t
int16_t	→ int32_t

How does one classify I/O port registers?
- Formally: Global = public permanent
- Practically: private permanent

Can access

Does access

Examples of constants

Symbol

- #define

```
#define IRSlope 1195172
#define IROffset -1058
```

ROM

- const

```
int32_t const ADCBuffer[16]=
{2000, 2733, 3466, 4199, 4932,
5665, 6398, 7131, 7864, 8597,
9330, 10063, 10796, 11529,
12262, 12995};
```

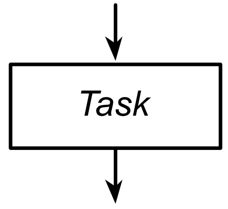
Enumerated types

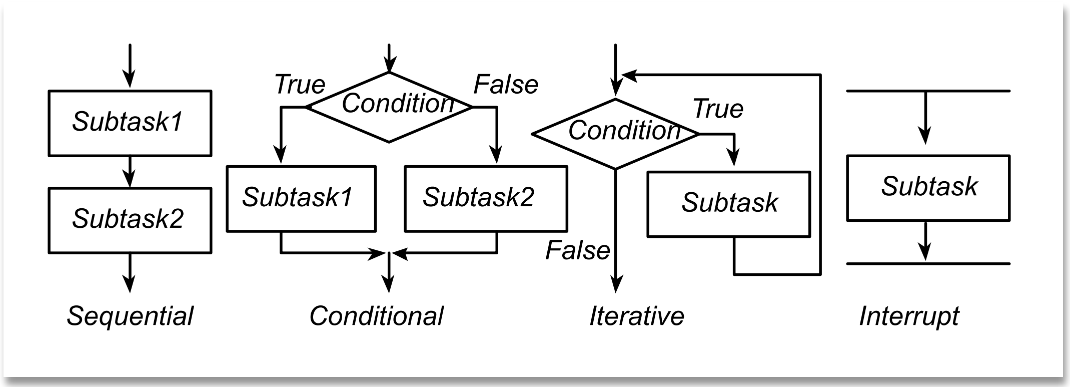
- enum

```
enum scenario {
  Error = 0,
  LeftTooClose = 1,
  RightTooClose = 2,
  CenterTooClose = 4,
};
typedef enum scenario scenario_t;
```

Software design, building blocks

- “do A then do B” → sequential
- “do A and B in either order” → sequential (parallel)
- “if A, then do B” → conditional
- “for each A, do B” → iterative
- “do A until B” → iterative
- “repeat A over & over forever” → iterative (condition always true)
- “on external event do B” → interrupt
- “every t msec do B” → interrupt

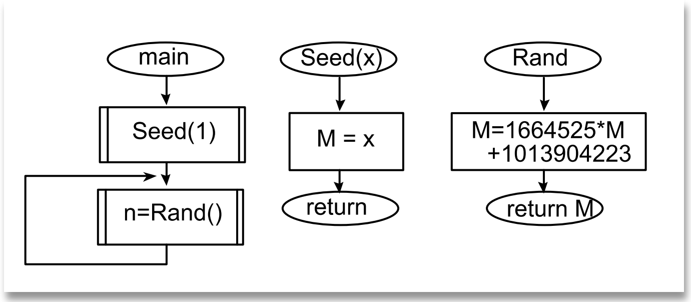


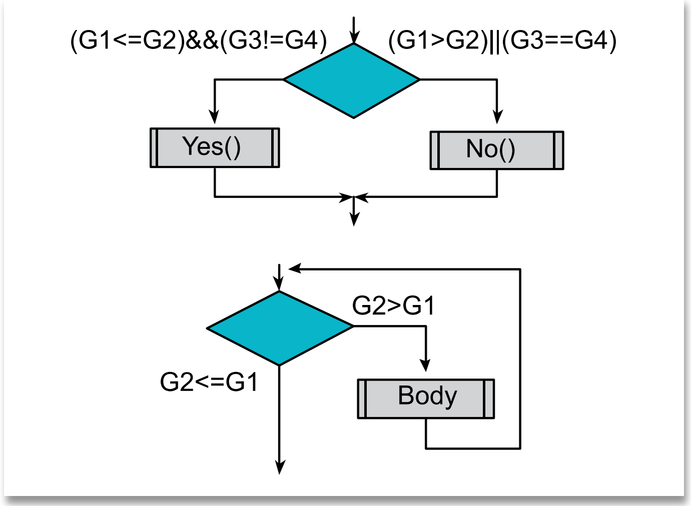


C Programming using MSP432

Summary

- Review C programming
  - Logic/shift operations
  - Arithmetic calculations
  - Functions
  - Conditionals
  - Variables
  - Constants





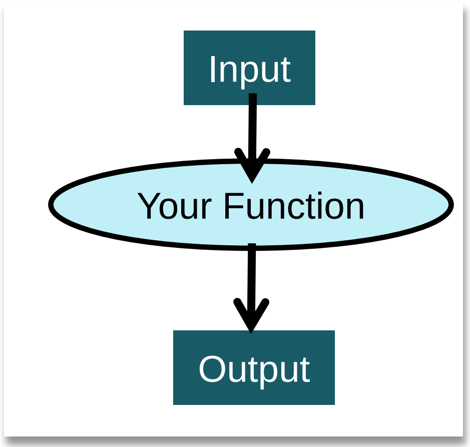
Module 4

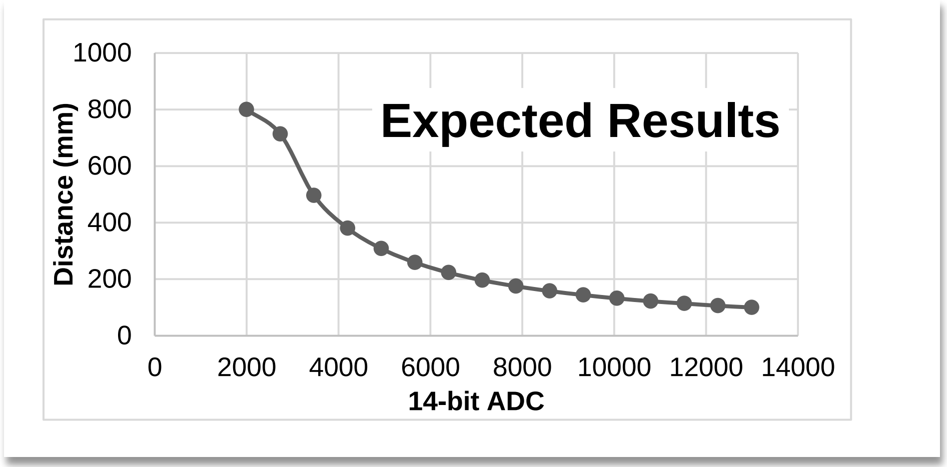
Lecture: Software Design using MSP432- Debugging

Debugging on the MSP432

You will learn in this module

- Debugging
  - Control (step, breakpoints)
  - Observing variables
  - Functional debugging





Debugging

- Functional Debugging
  - Known inputs
  - Expected outputs
- Stabilization
  - Fix input values, fix timing of input
  - Repeated testing shows changes in software
- Test cases
  - Near the extremes and in the middle
  - Most typical of how clients will properly use the system
  - Most typical of how clients will improperly use the system
  - That differ by one
  - You know your system will find difficult (corner cases)
  - Using a random number generator

Important aspects:

- Control
- Observability

```
// Program 4_1 used to test the Convert function
int32_t const ADCBuffer[16]={2000,2733,3466,4199,4932,
 5665, 6398, 7131, 7864, 8597, 9330, 10063, 10796,
 11529, 12262, 12995};
int32_t const DistanceBuffer[16]={800,713,496,380,
 308,259,223,196,175,158,144,132,122,114,106,100};
void Program4_1(void){int i;
int32_t adc,distance,errors,diff;
  errors = 0;
  for(i=0; i<16; i++){
    adc = ADCBuffer[i];
    distance = Convert(adc); // call to your function
    diff = distance-DistanceBuffer[i];
    if((diff<-1)||(diff>1)){
      errors++;
    }
  }
  while(1){};
}
```

Debugging (Control)

- Test cases
  - Get data from arrays (rather than actual inputs devices)
- Single step
  - Step, step over, step in, step out
- Breakpoints
  - Set using debugger
- Special test main
  - Establish exact scenario you wish to test
  - Stabilization

Important aspects:

- Control
- Observability

```
int32_t errors;
void Program4_2(void){
  scenario_t result,truth;
  int i,j,k;
  int32_t left, right, center; // sensor readings
  errors = 0;
  for(i=0; i<18; i++){
    left = CornerCases[i];
    for(j=0; j<18; j++){
      center = CornerCases[j];
      for(k=0; k<18; k++){
        right = CornerCases[k];
        result = Classify(left,center,right);
        truth = Solution(left,center,right);
        if(result != truth){
          errors++;
        }
      }
    }
  }
  while(1){
  }
}
```

Debugging (Observability)

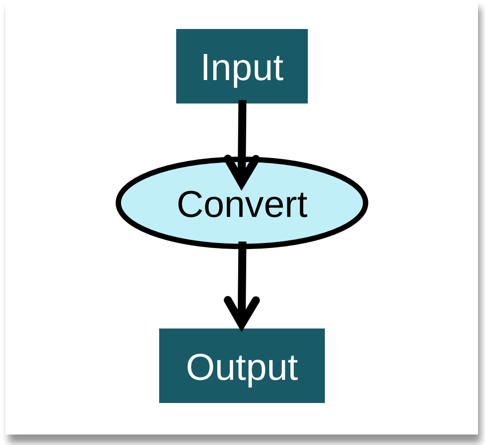
- Debugger monitor windows
  - Globals
  - Locals
  - I/O registers
- Dump
  - Save results in RAM or ROM
- Output to UART
  - Observe with terminal program like PuTTY or TExaSdisplay
- Hardware Monitors
  - Lights, sounds
  - Nokia 5110 LCD display

Important aspects:
- Control
- Observability

Debugging on the MSP432

Summary

- Debugging
  - Control
  - Observability
  - Functional debugging


Input
Convert
Output