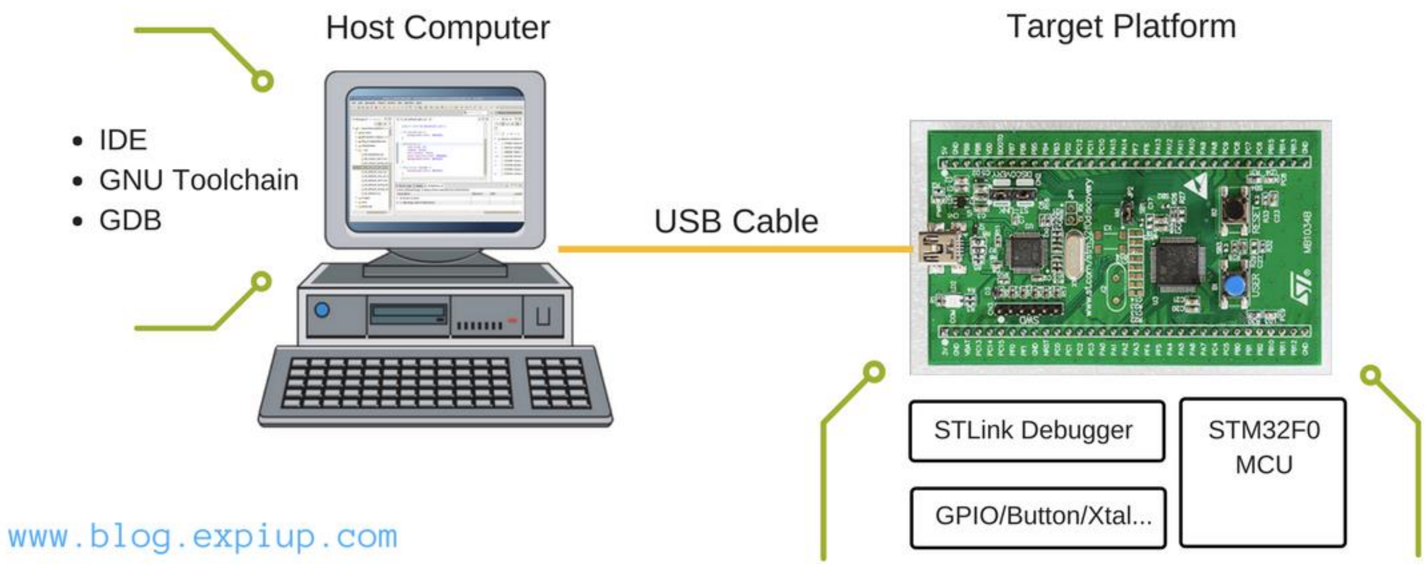
CO3053 – Embedded Systems

5. Embedded Programming Paradigm



- Round Robin
- Round Robin with Interrupt
  - Event-Driven and Time-Driven
- Real Time Operating System

Contents

Round Robin

- Simplest architecture, a single loop checks devices in predefined sequence and performs I/O right away

```
while(1) {
   if (device_1_ready()) {  /*Perform D1 I/O and relate computation.*/ }
   if (device_2_ready()) {  /*Perform D2 I/O and relate computation.*/ }
   …
   if (device_N_ready()) {  /*Perform DN I/O and relate computation.*/ }
}
```

- Works well for system with few devices, trivial timing constraints, proportionally small processing costs
- Response time of device *i* equal to WCET (Worst Case of Execution Time) of the body of the loop

Round Robin

- Periodic Round Robin
  - In case the system must perform operations at different frequencies
  - Add code to wait a variable amount of time

- Exercise
  - Think how to implement a loop that runs every 10ms and measures the drift

```
while(1) {
    waitForNextPeriod(10);   // idle for up to 10 ms
    if (device_1_ready()) {  /*Perform D1 I/O and relate computation.*/ }
    ...
```

Round Robin

- Limitations
  - If some devices require small response times, while other have large WCET it will not be possible to guarantee that all timing constraints will be met.
  - The architecture is fragile, adding a new task can easily cause missed deadlines.

- Question
  - Is the order in which devices appear significant?
  - Same above question, but with code for devices having different processing times and timing constraints?

Round Robin with Interrupts

- Hardware events requiring small response times handled by ISRs
- Typically ISRs do little more than set flags and copy data

```
bool f_device_1 = FALSE;
bool f_device_1 = FALSE;
       …
void interrupt handle_dev_1() {
    // handle device 1
    f_device_1 = TRUE;
}
         …
void main() {
  while (1) {
     if(f_device_1) {
       f_device_1 = FALSE;
       // do processing related to device 1…
     if (f_device_2) {
          …
  }
}
```

Round Robin with Interrupts

- Interrupt routines deal with the very urgent needs of devices.
  - Non-urgent tasks are executed in a robin-round fashion
  - Interrupt can be Time-driven or Event-driven
- Interrupt routines set flags to indicate the interrupt happened.
  - Urgent tasks can be prioritized
- Drawbacks
  - Shared-data problems arise
  - Time response for a non-urgent task
    - duration of the main loop + interrupts

Round Robin with Interrupts

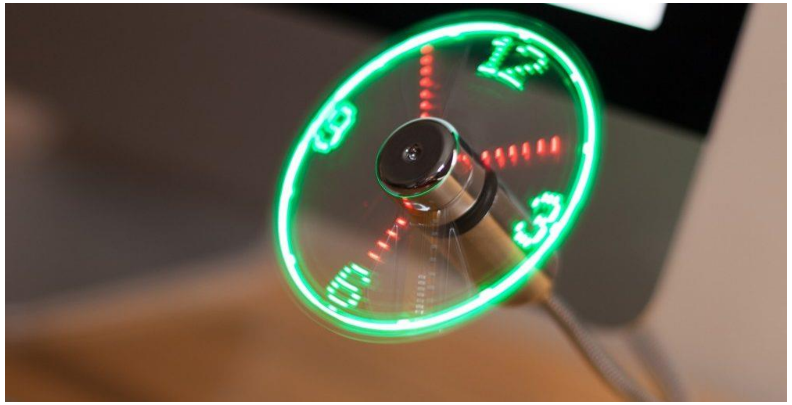
```
volatile BOOL ready1 = 0, ...,
              readyn = 0;

interrupt void urgent1(void){
    !! urgent operations of task 1;
    ready1 = 1;
    }
                 ⋮
interrupt void urgentn(void){
    !! urgent operations of task n;
    readyn = 1;
    }
```

```
void main(void){
    while (TRUE){
        if (ready1){
            !! non-urgent operations of task1;
            ready1 = 0;
            }
                          ⋮
        if (readyn){
            !! non-urgent operations of taskn;
            readyn = 0;
            }
    }
}
```

Shared-data problems

Round Robin with Interrupts

- Example
  - Propeller clock



```
volatile BOOL next_image = 0;

interrupt void Timer0(void){
    !! Update current pixel line;
}
interrupt void CompleteRev(void){
    !! Update image;
    next_image = 1;
}
```

```
void main(void){
    init();
    while (TRUE){
        If (next_image){
            !! Compute next image
            next_image = 0;
        }
        !! Check switches
        !! Select image to display
    }
}
```

Round Robin with Interrupts

- Questions
  - What if all task code executes at same priority?
  - What if one of the device requires large amount of processing time (larger than the time constraint of others)?

Real-Time Operating System (RTOS)

- An RTOS is an OS for response time-controlled and event-controlled processes.
- It is very essential for large-scale embedded systems.
- The main task of a RTOS is to manage the resources of the system such that a particular operation executes in precisely the same amount of time every time it occurs

Real-Time Operating System (RTOS)

- Interrupt routines execute urgent tasks and signal that non-urgent tasks are ready to be executed.
- The operating system invokes dynamically the non-urgent tasks.
- The OS is able to suspend the execution of a task to allow another one to be executed. (Preemptive Scheduling Support)
- The OS handles communication between tasks.

```
#include "signal.h”

interrupt void urgent1(void){
    !! urgent operations of task 1;
    !! send signal 1;
}

                  ⋮

interrupt void urgentn(void){
    !! urgent operations of task n;
    !! send signal n;
}
```

Real-Time Operating System (RTOS)

- Data communication between tasks/interrupts must be coordinated
- Complex implementation (but you don't have to do it yourself)
- Robustness against modifications
- The OS uses a certain portion of the processor resources (2% to 4%)

```
void task1(void){
    !! wait for signal 1;
    !! non-urgent operations of task 1;
}

                    :

void taskn(void){
    !! wait for signal n;
    !! non-urgent operations of task n;
}

void main(void){
    !! initialize the operating system;
    !! create and enable tasks;
    !! start task sequencing;
}
```

Selection Strategy

- We want to obtain the greatest amount of control over the system response time ➠ Select the simplest architecture that will meet your response requirements.
- RTOSs should be used where response requirements demand them.

Discussion

- Simple video game (such as PONG)

What has to be considered?

  - Display the image (PAL signal: 625 lines @ 50Hz)
  - Game management (i.e. compute the position of the ball)
  - Game control (buttons, controller)

Discussion

- Vending Machine

What has to be considered?

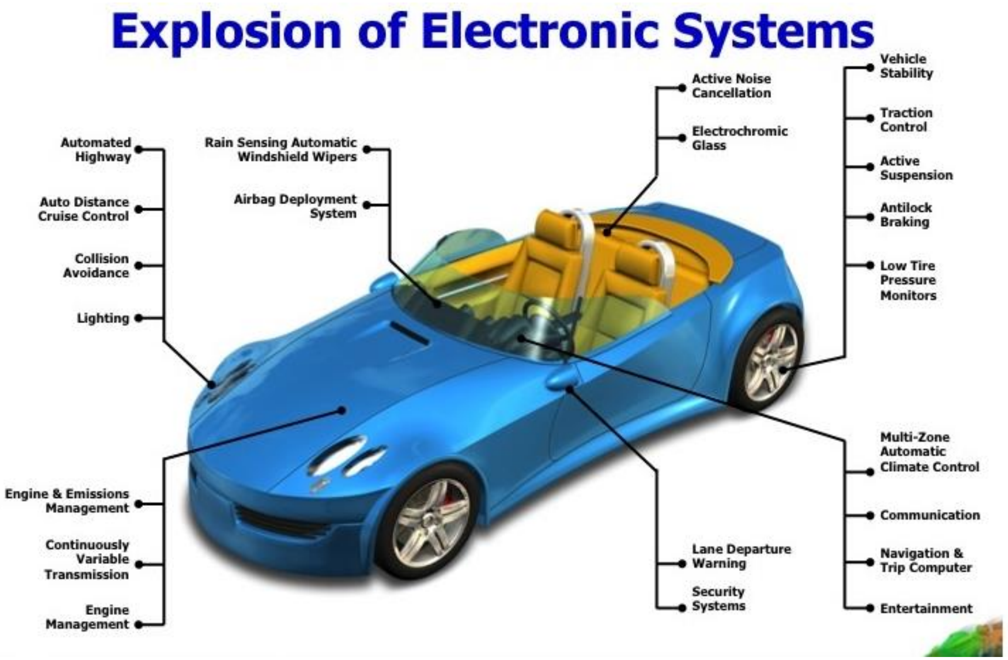
- Display information
- Handle buttons & coin acceptor
- Check sensors
- Motors control

Discussion

- Vehicle embedded electronics

What has to be considered?

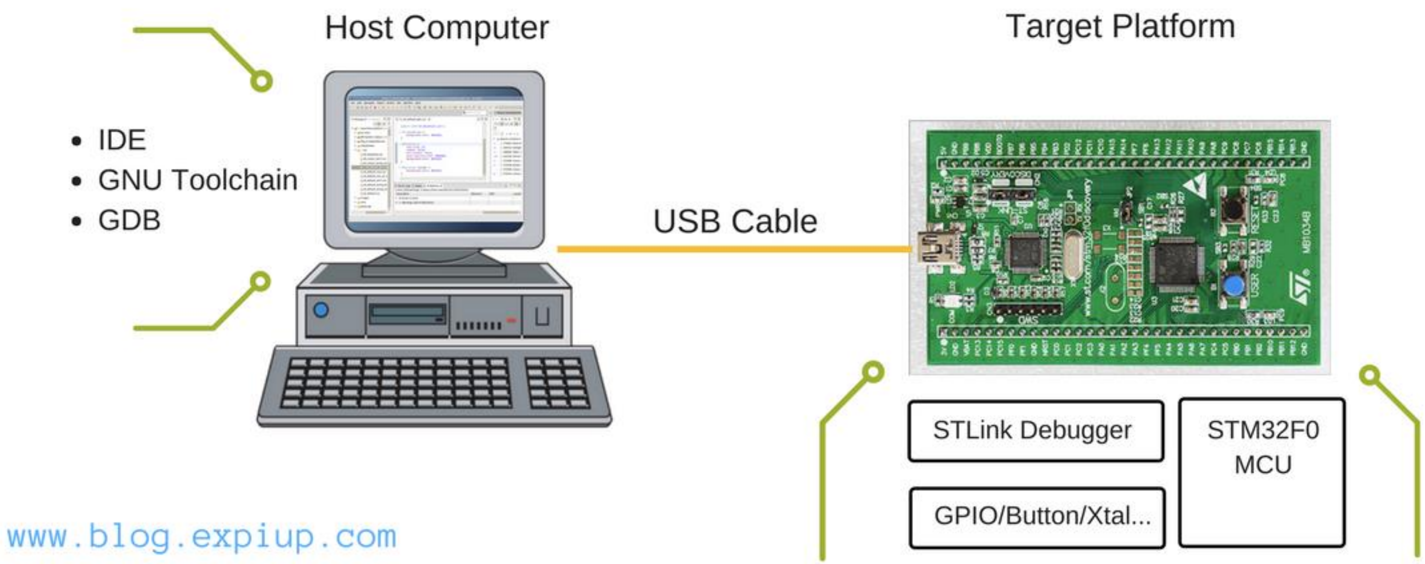
  - Sensor measurement (pedal, speed, switches, …)
  - Engine control (ignition, turbo, injection, cooling system, …)
  - Cruise-control
  - Display
  - GPS



Reference and Further Readings

- https://lectures.tik.ee.ethz.ch/es/slides/4_ProgrammingParadigms.pdf

Miscellaneous Topics for Efficient C Programming



Problems with #define

```
#define PI_PLUS_1        3.14 + 1
.
.
x = 5 * PI_PLUS_1;    // The compiler sees the statement as
                      // x = 5 * 3.14 + 1
                      //So, it will be resolved as follows:
                      // x = (5 * 3.14) + 1
                      //Which is not what we want!
                      // Solution: #define PI_PLUS_1  (3.14 + 1)
                      //Moral: Beware of the "()" while dealing
                      //with the #define statement
```

Problem with Macros (1)

```
#define ADD(a,b) a + b
.
.
c = 2 + ADD(1,2);    // Result is 5 → Correct

d = 2 * ADD(1,2);    // Result is 4 → Incorrect
```

```
#define ADD(a,b) (a + b)
.
.
c = 2 * ADD(1,2);    // Result is 6 → Correct
```

Moral: Again, beware of the "()" while dealing with the #define statement

Problem with Macros (2)

```
#define MULT(a,b)  (a * b)
.
.
c = 3 + MULT(1,2);          // Result is 5 → Correct

d = 3 + MULT(1+1,2+2);      // Result is 8 → Incorrect
```

```
#define MULT(a,b)  ((a) * (b))
.
.
d = 3 + MULT(1+1,2+2);      // Result is 11 → Correct
```

Moral: I told you! Beware of the "()" while dealing with the #define statement

Playing around with Increment

- Example 1:
  a = 2;
  b = a++;
  //Values after: a = 3, while b = 2

- Example 2:
  a = 2;
  b = ++a;
  //Values after: a = 3, while b = 3

- Example 3:
  a = 5;
  b = 2;
  c = a+++b;
  //Values after: a = 6, b = 2, while c = 7

Bit Manipulation (1)

- Detect if two integers have opposite signs:

```
int x, y;              // input values to compare signs bool
f = ((x ^ y) < 0);     // true iff x and y have opposite signs
```

- Determine if an unsigned integer is zero or a power of 2:

```
unsigned int v;        // we want to see if v is zero or a power of 2
bool f;                // the result goes here
f = (v & (v - 1)) == 0;
```

- Determine if an unsigned integer is a power of 2:

```
f = v && !(v & (v - 1));
```

Bit Manipulation (2)

- Merge bits from two values according to a mask:

```
unsigned int a;     // value to merge in non-masked bits unsigned
int b;              // value to merge in masked bits unsigned
int mask;         // 1 where bits from b should be selected; 0 where from a.
unsigned int r;     // result of (a & ~mask) | (b & mask) goes here
r = a ^ ((a ^ b) & mask);
```

- Counting bits set:

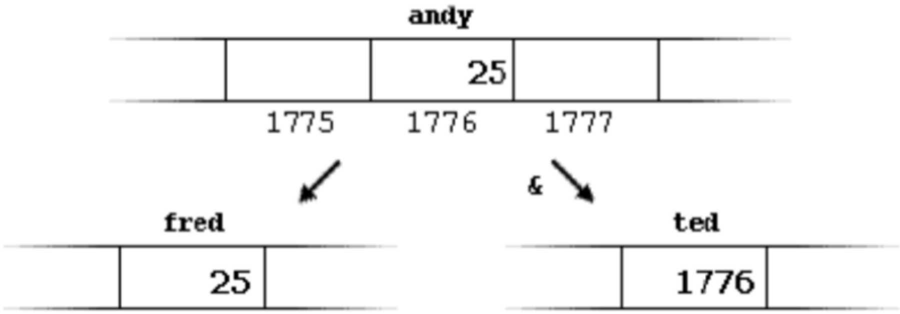
```
unsigned int v; // count the number of bits set in v
unsigned int c; // c accumulates the total bits set in v
for (c = 0; v; v >>= 1)
        c += v & 1;
```

Pointers

- Reference to a data object or a function
- Helpful for "call-by-reference" functions and dynamic data structures implementations
- Very often the only efficient way to manage large volumes of data is to manipulate not the data itself, but pointers to the data

Example:

```
andy = 25;
fred = andy;
ted = &andy;
```



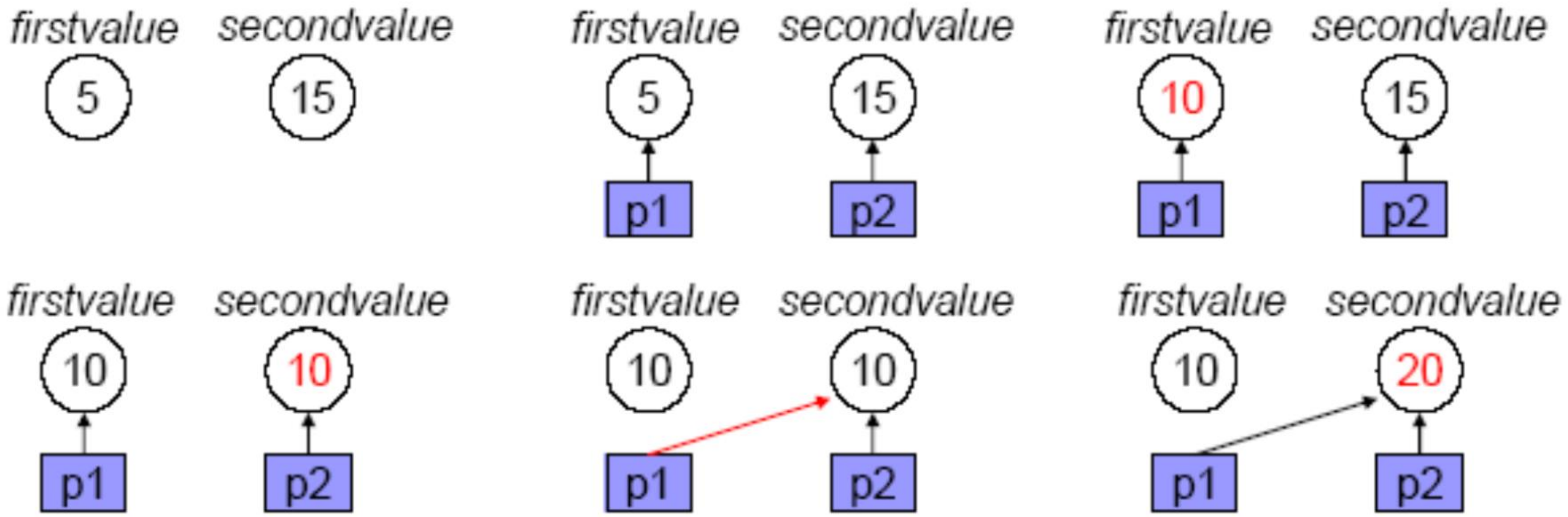
Pointers Example

```
int firstvalue = 5, secondvalue = 15;
int * p1, * p2;

p1 = &firstvalue;  // p1 = address of firstvalue
p2 = &secondvalue; // p2 = address of secondvalue
*p1 = 10;          // value pointed by p1 = 10
*p2 = *p1;         // value pointed by p2 = value pointed by p1
p1 = p2;           // p1 = p2 (value of pointer is copied)
*p1 = 20;          // value pointed by p1 = 20
```

firstvalue = ?
secondvalue = ?

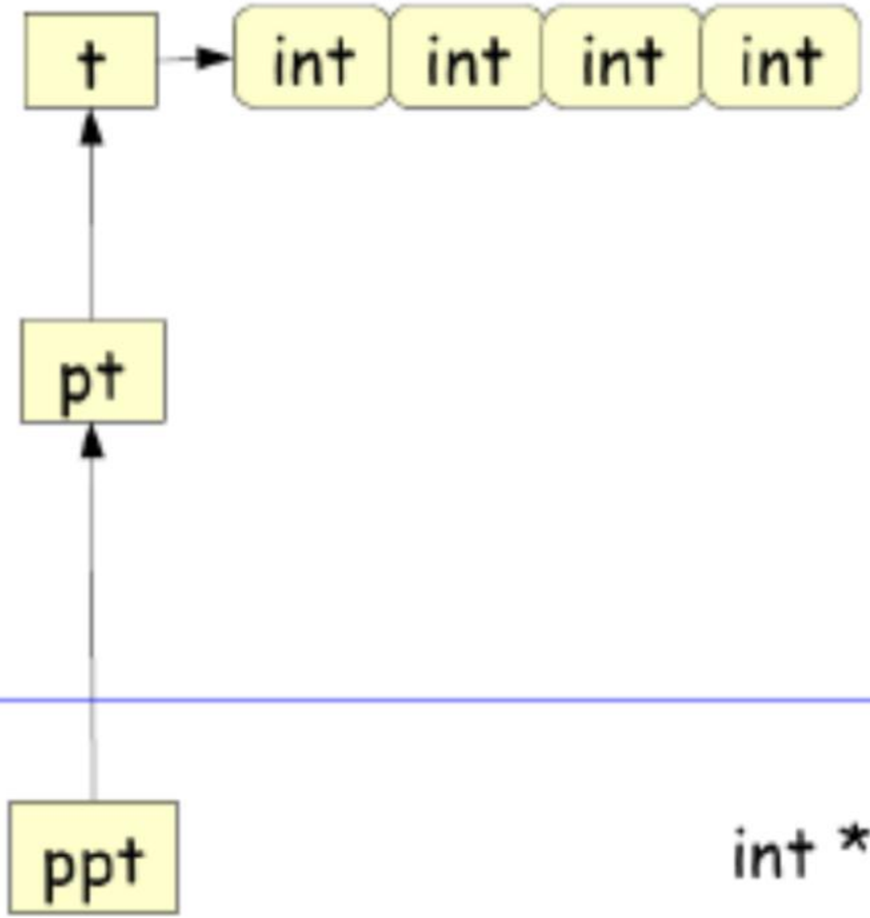
Pointers Example

```
int firstvalue = 5, secondvalue = 15;
int * p1, * p2;

p1 = &firstvalue;  // p1 = address of firstvalue
p2 = &secondvalue; // p2 = address of secondvalue
*p1 = 10;          // value pointed by p1 = 10
*p2 = *p1;         // value pointed by p2 = value pointed by p1
p1 = p2;           // p1 = p2 (value of pointer is copied)
*p1 = 20;          // value pointed by p1 = 20
```



More Pointers Fun

```
int table[4];
int *t = table;
```



int * : points to one or more ints (table of ints).

```
int **pt = &t;
```

int ** : points to one or more int* (table of int*).

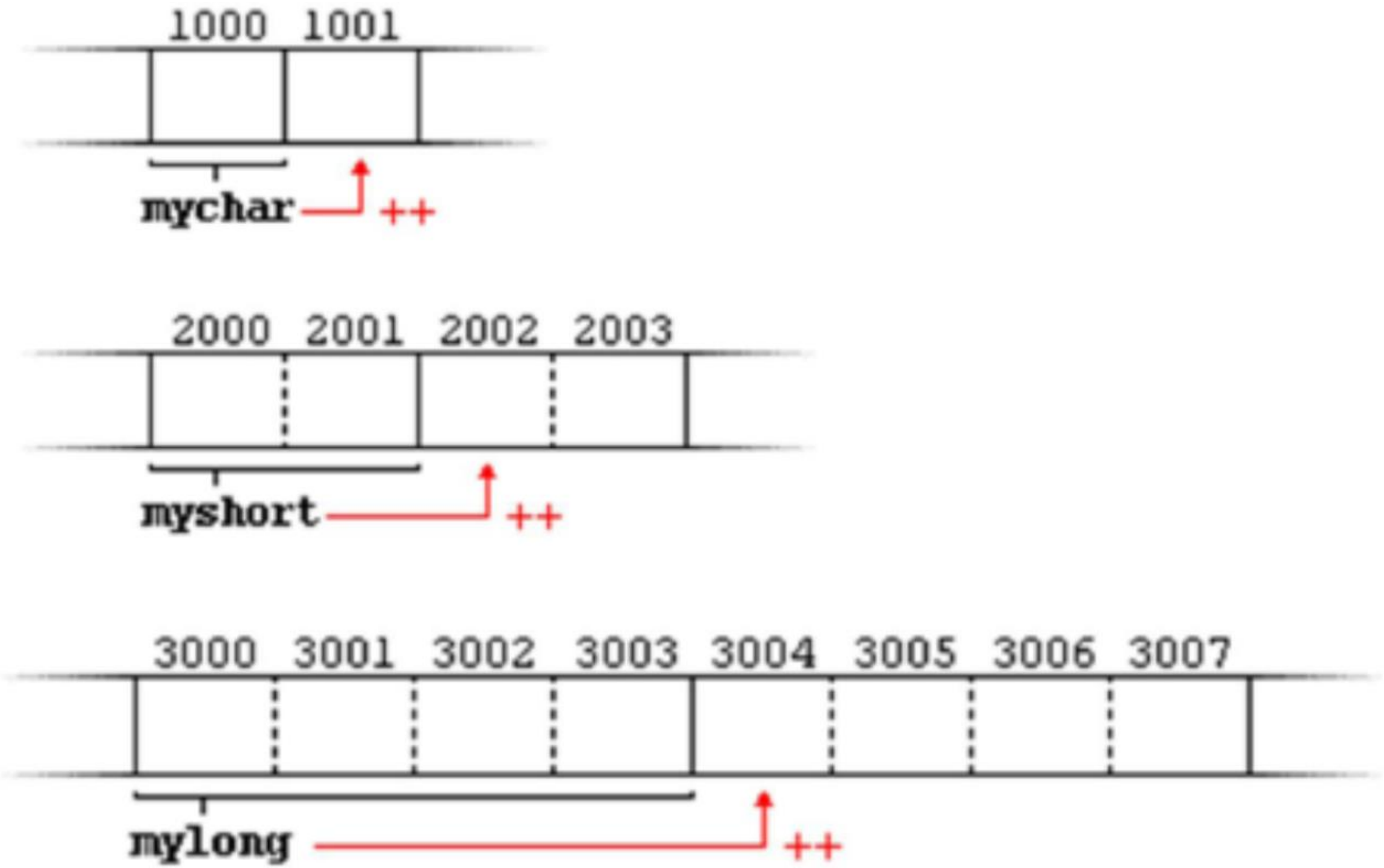
```
int ***ppt = &pt;
```

int *** : ... you get it now!

Pointers are Typed

```
char *mychar;
short *myshort;
long *mylong;
```

```
mychar++;
myshort++;
mylong++;
```

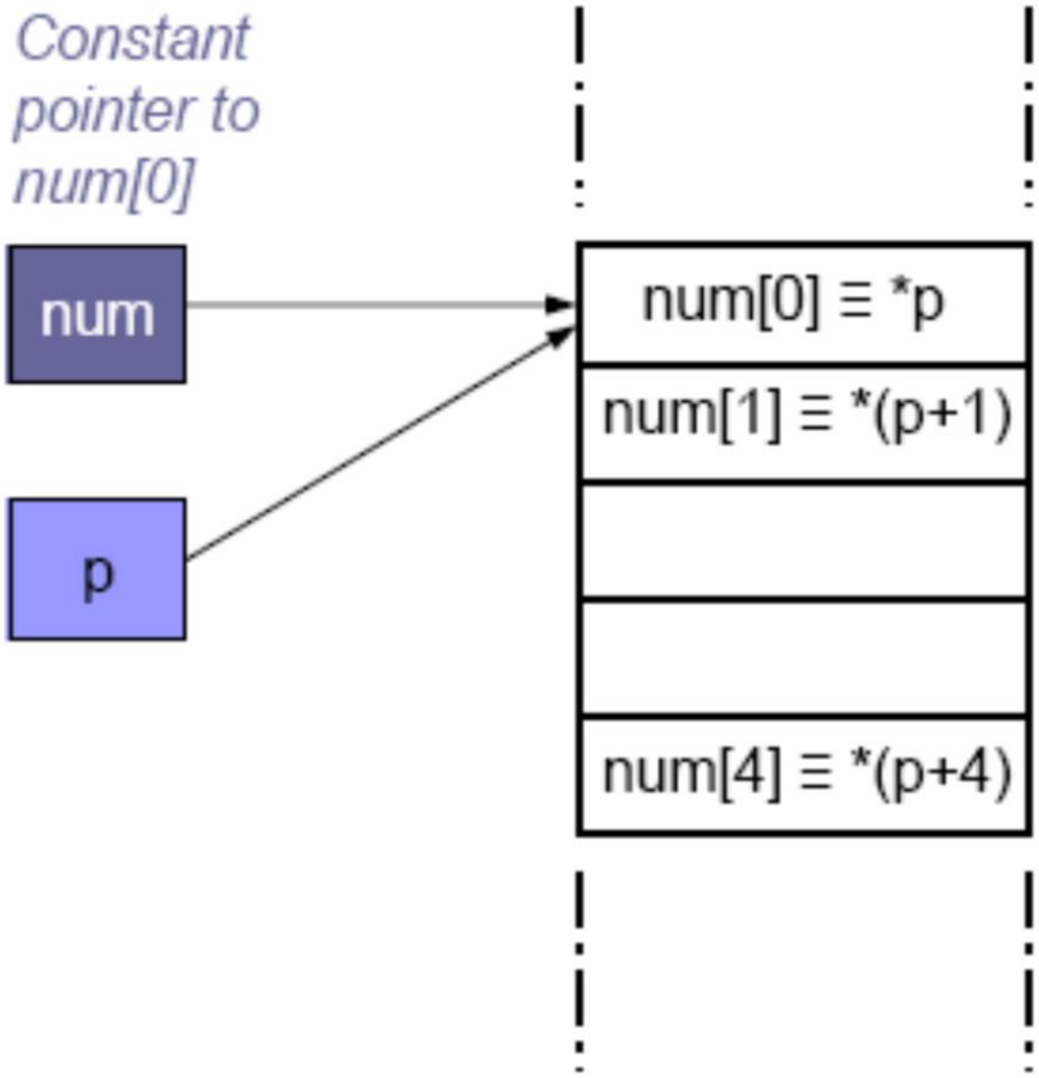


Pointers and Array

- *int num[5];*
- *int *p;*
- *p = num;*

~~*num = p;*~~

An array is a constant pointer



Pointers Precedence Issues

- *(array+i) ≡ array[i]
- *array+i ≡ array[0] + i
- *p++ ≡ *(p++)
- Notice the difference with (*p)++
- Better use parentheses to prevent mistakes
- int * ptr1, ptr2; Vs int * ptr1, * ptr2;

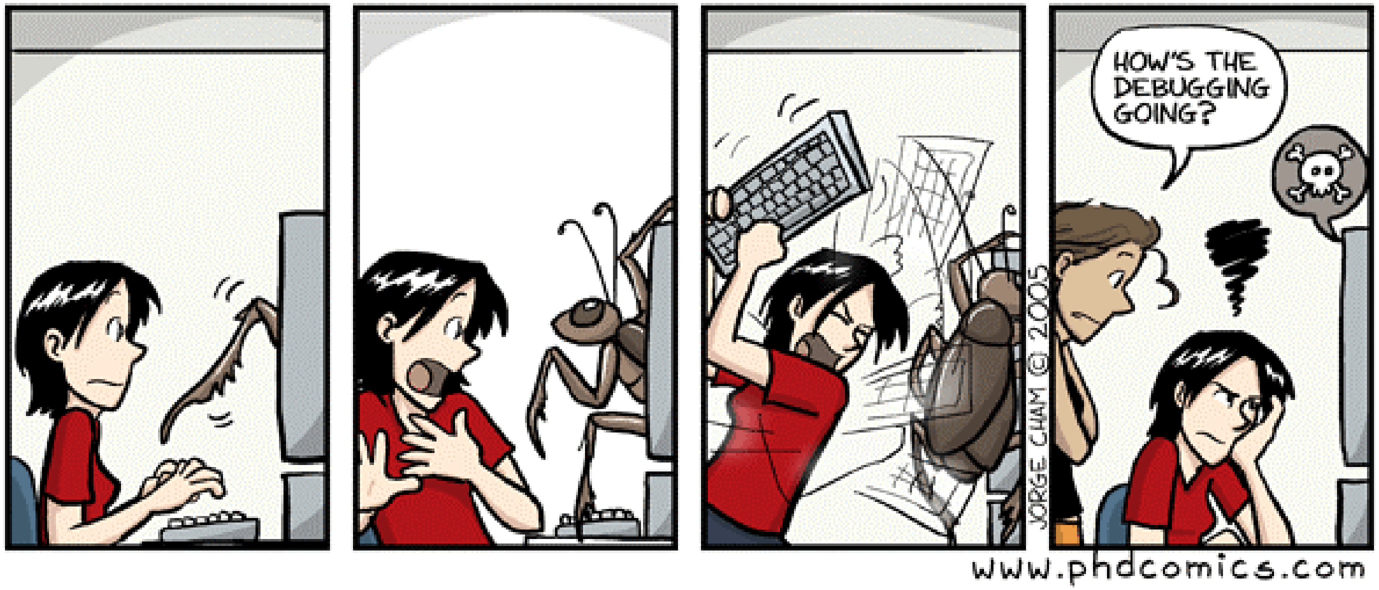
Efficient C Programming

- How to write C code in a style that will compile efficiently (**increased speed and reduced code size**) on ARM architecture?
  - How to use data types efficiently?
  - How to write loops efficiently?
  - How to allocate important variables to registers?
  - How to reduce the overhead of a function call?
  - How to pack data and access memory efficiently?

References

- A.N. Sloss, D. Symes, and C. Wright, "ARM System Developers Guide"

Debugging



9 Indispensable Rules for Finding the Most Elusive Software and Hardware Problems

David J. Agans

Debugging

9 Indispensable Rules for Finding the Most Elusive Software and Hardware Problems

David J. Agans

Rule #1 – Understand the System

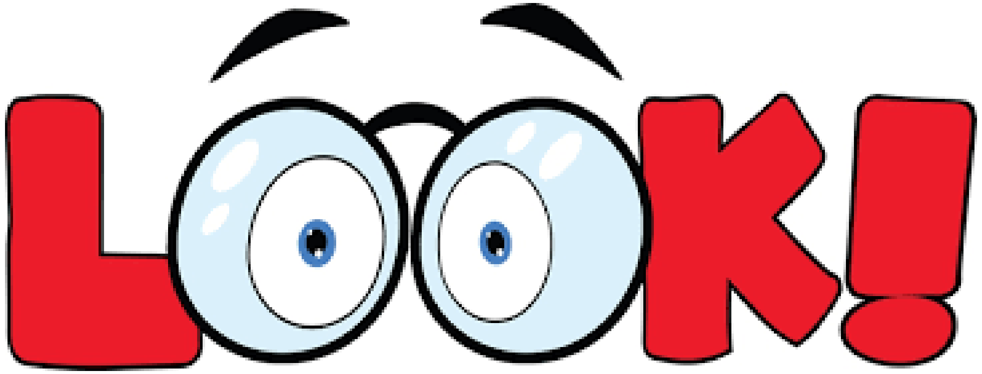
- Read the manual (datasheet).
- Debugging something you don't understand is pointlessly hard.
- Just as with testing, subject knowledge matters – here you need knowledge of the source code as well.

Rule #2 – Make It Fail

- You cant debug what you cant produce.
- Find a way to reliably make a system fail.
- Record everything, and look for correlation.

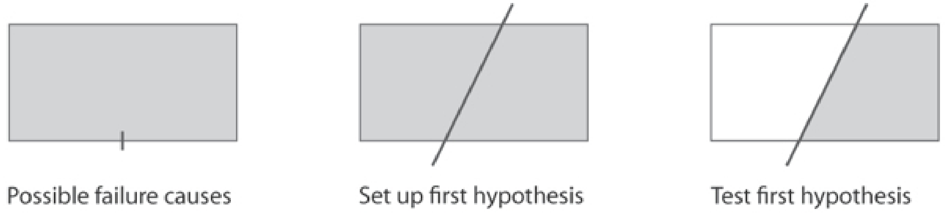
Rule #3 – Quit Thinking and Look

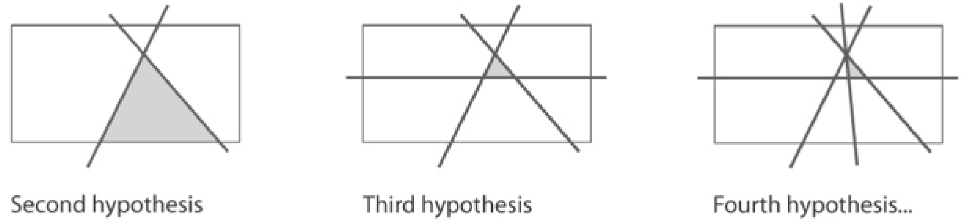
- Don’t hypothesize before examining the failure in detail
  - Examine the evidence, then think.
- Engineers like to think, don’t like to look nearly as much.
- If it is doing X, must be Y – maybe
  - Check



Rule #4 – Divide and Conquer

- **This rule is the heart of debugging**
  - Delta-debugging





  - Narrow down the source of the problem
  - Does it still fail if this factor is removed?
  - Use a debugger to check system state at checkpoints; if everything is ok, you are before the problem.

Rule #5 – Change One Thing at a Time

- A common very bad debugging strategy
  - It could be one of X, Y, X.
  - I shall change all three and run it again.

- Isolate factors, because that is how you get experiments that tell you something.

- If **code worked before last check-in**, maybe you should look at just those changes.

Rule #6 – Keep an Audit Trail

- Don't rely on your perfect memory to remember everything you tried.

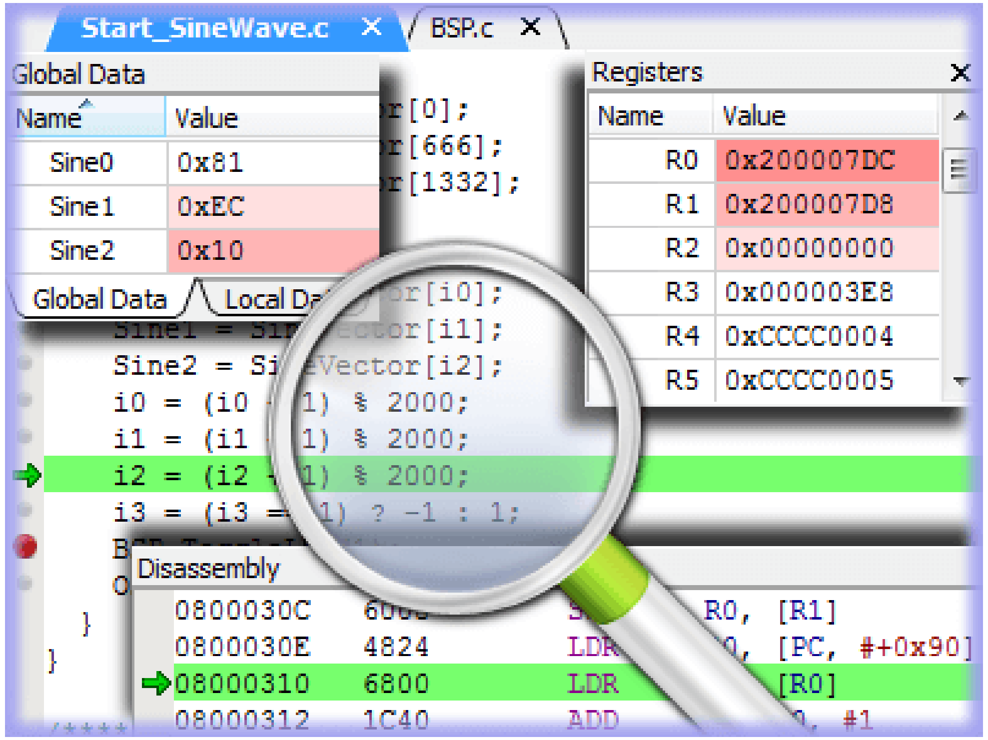
- Don't assume only you will ever work on this problem.

Rule #7 – Check the Plug

- Question assumptions
  - Are you running the right code?
  - Are you out of gas?
  - Is it plugged in?

- Start at the beginning
  - Did you initialize memory properly?
  - Did you turn it on?

- Test the tool
  - Are you running the right compiler?
  - Does the meter have a dead battery?



Rule #8 – Get a Fresh View

- Experts can be useful

- **Explain what happens**, NOT what you think is going on

Rule #9 – If you didn't Fix it, It ain't Fixed

- Once you "find the cause of a bug" confirm that changing the cause actually removes the effect.

- A bug is not done until the fix is in place and confirmed to actually fix the problem.
  - You might have just understood a symptom, not the underlying problem

Summary

1. Understand the system
2. Make It Fail
3. Quit Thinking and Look
4. Divide and Conquer
5. Change One Thing at a Time
6. Keep An Audit Trail
7. Check The Plug
8. Get A Fresh View
9. If You Didn’t Fix It, It ain’t Fixed