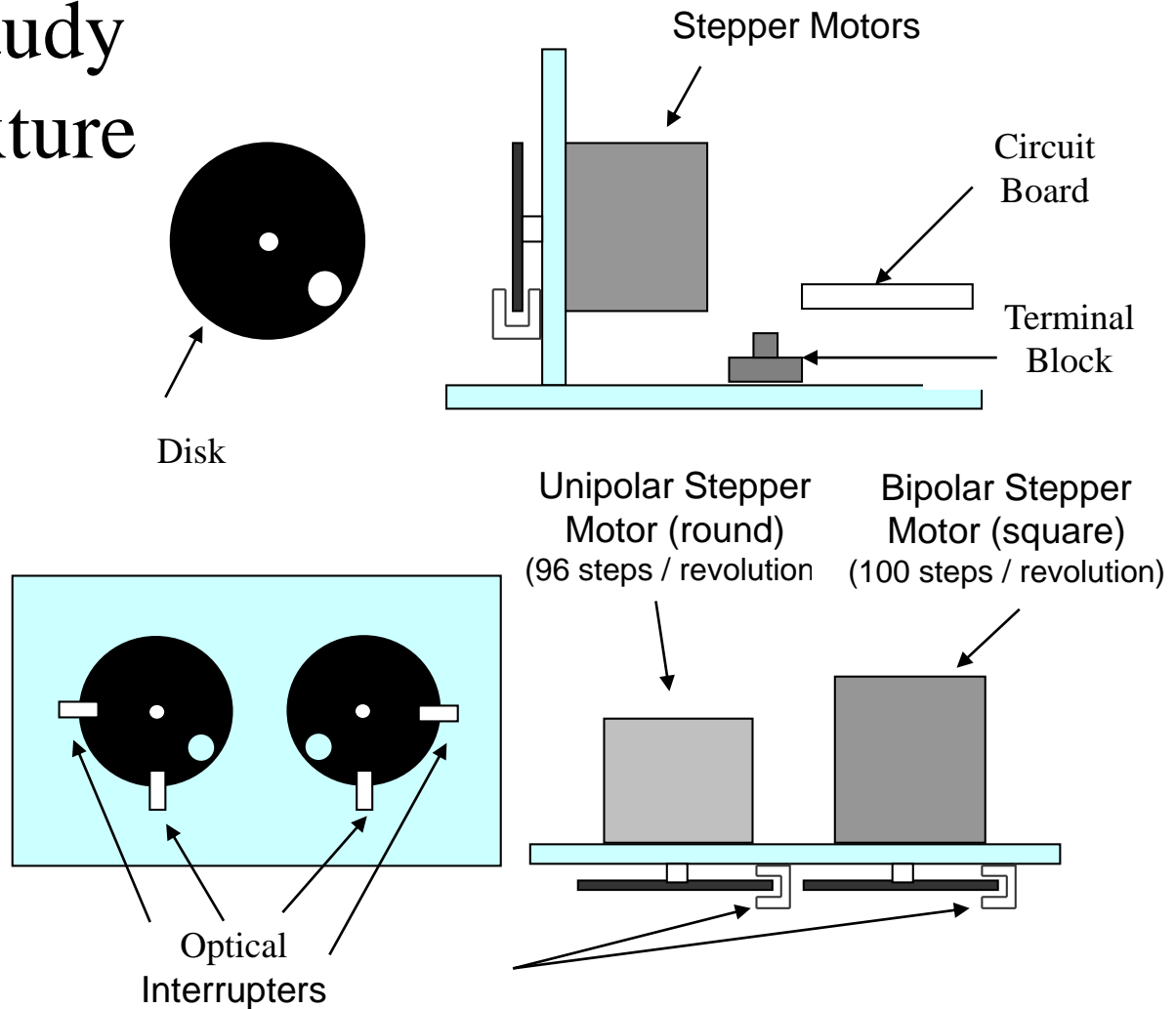


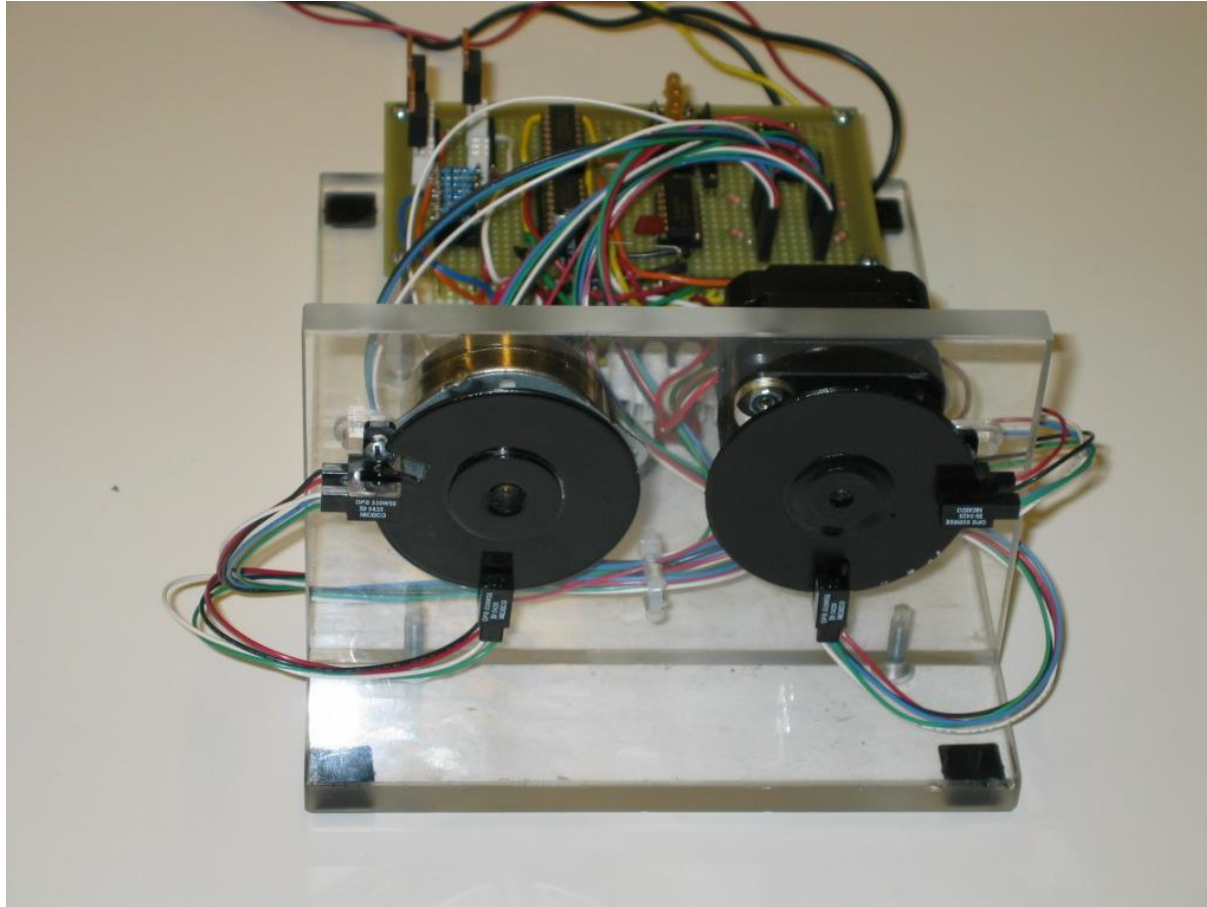
# State Machines

## Embedded C Programming

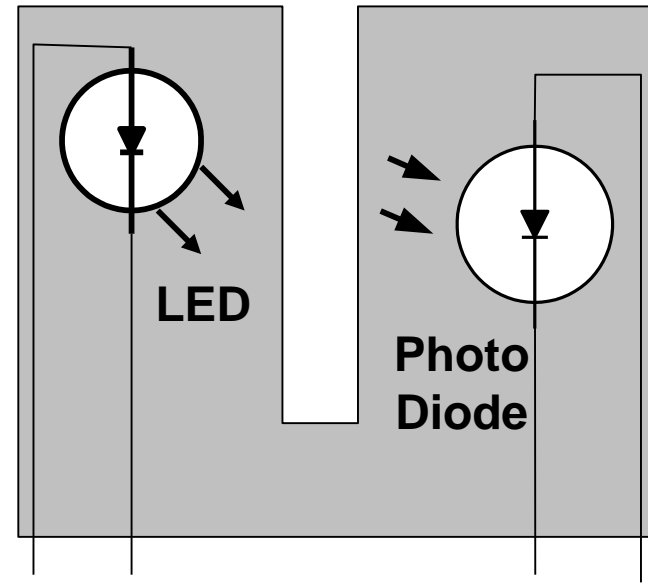
# Case Study Test Fixture



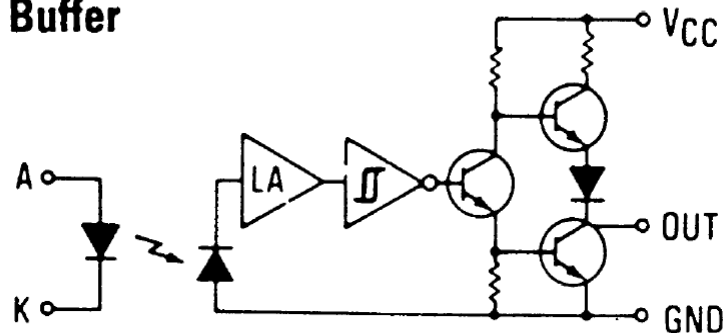
# Test Stand



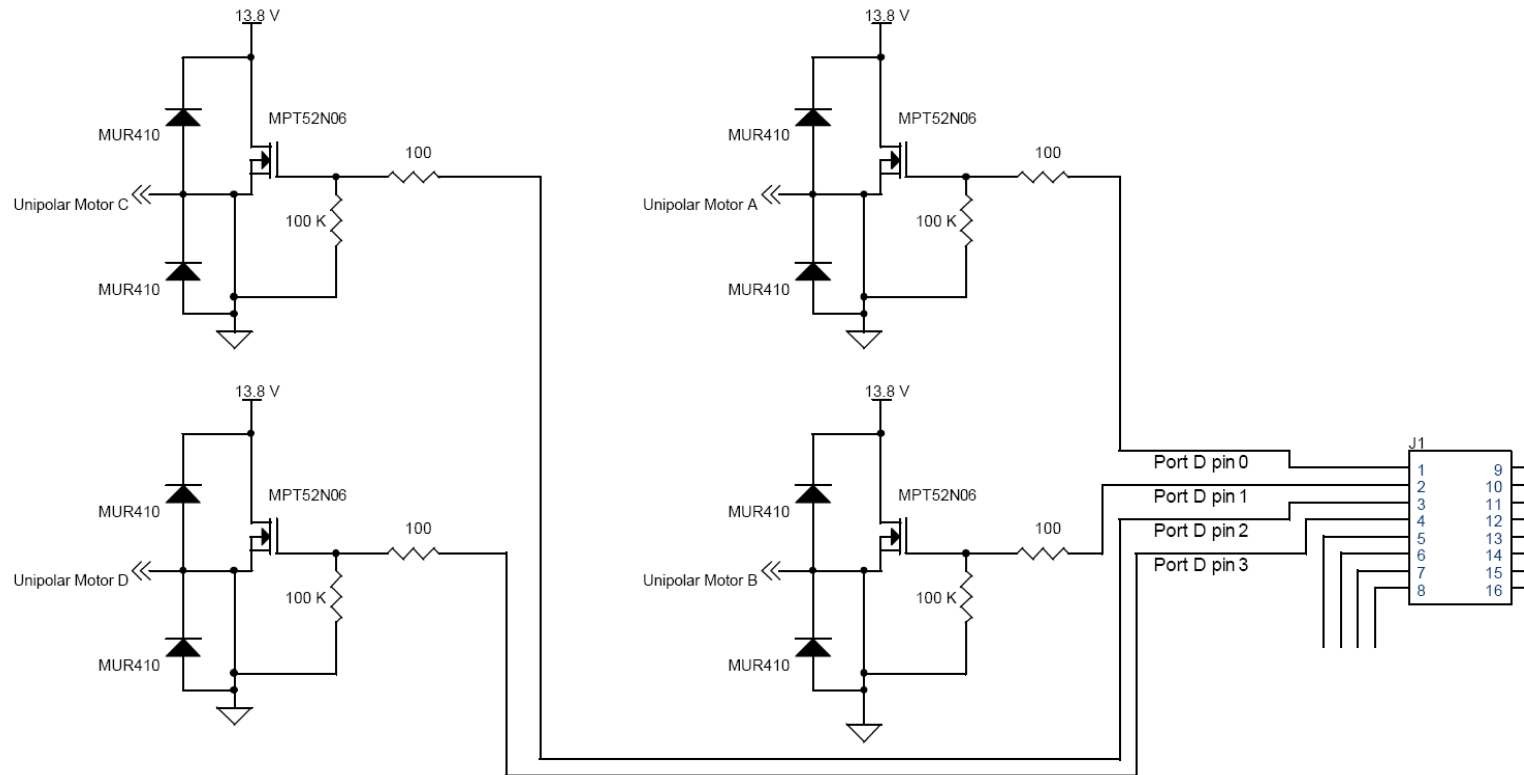
# Optical Interrupter



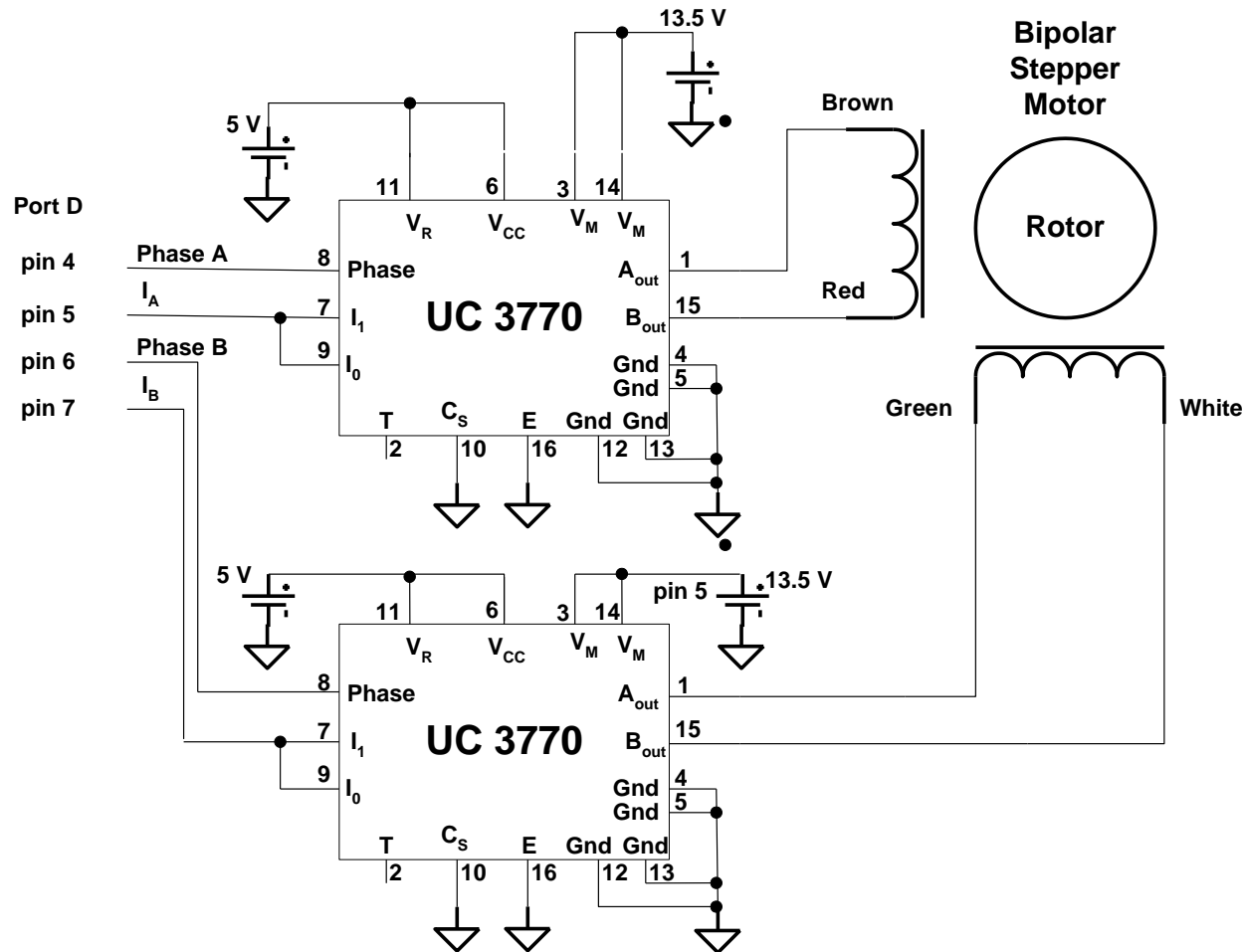
**OPB930, OPB940  
(Totem-Pole Output)  
Buffer**



# Unipolar Motor Driver



# Bipolar Stepper Motor Driver



all drivers  
should be off.

Reset

Initialize  
Ports & Registers

Synchronize Both Motors  
Bring Both Motors to Home

Green  
Press ?

No

Yes

GreenPress

Green  
Release ?

No

Yes

Read Mode  
Switch

Case Study  
Operation

Go To  
Mode

Mode 1

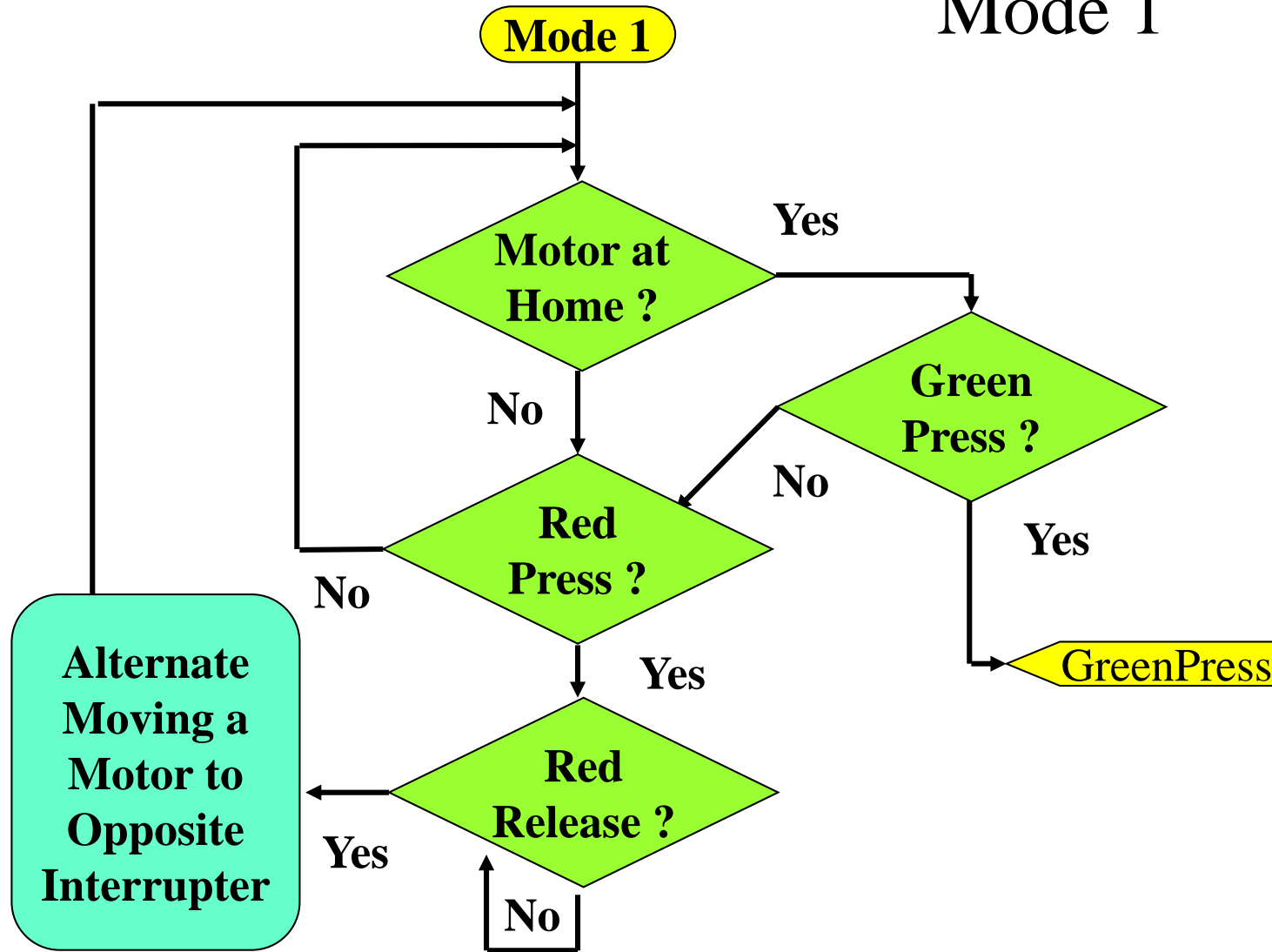
Mode 2

Mode 3

Mode 4

Error

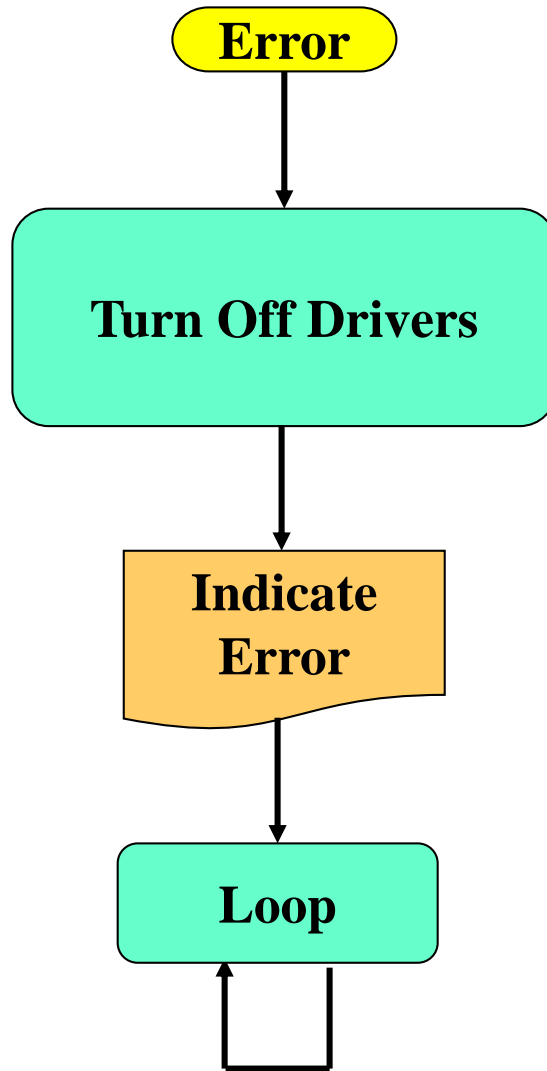
# Mode 1





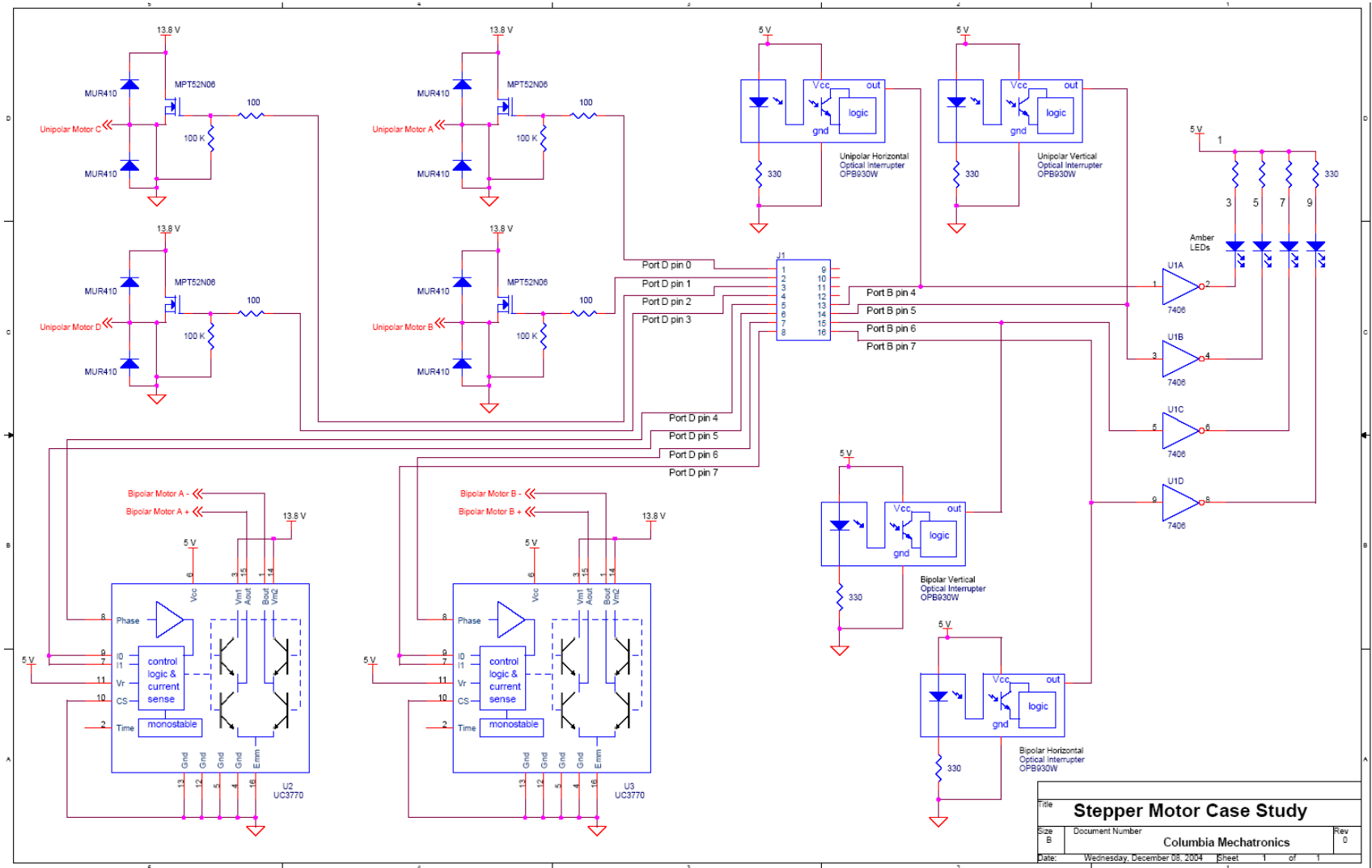


# Error

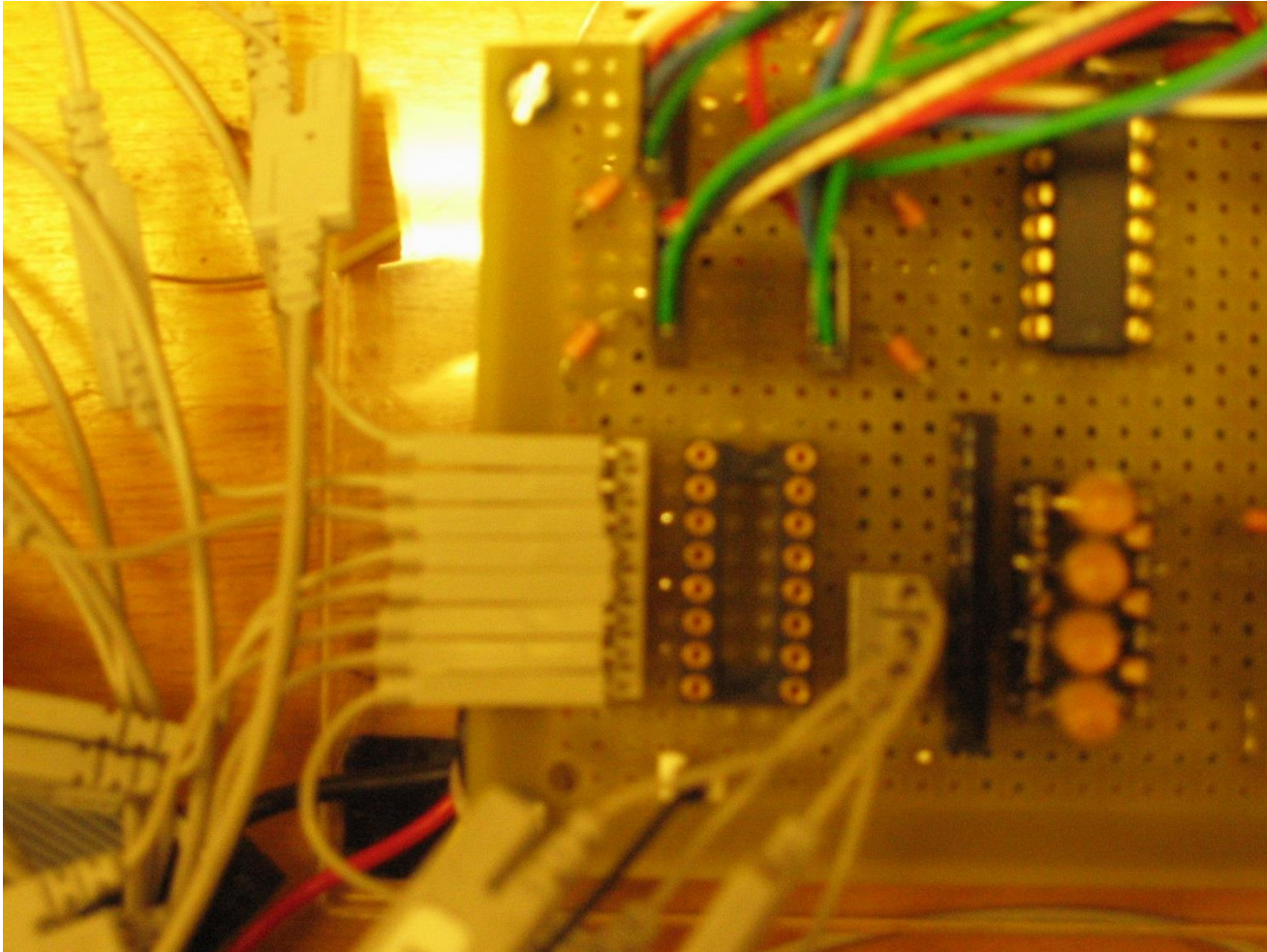


**After a fault, the processor has to be reset (or powered off and on) to continue.**

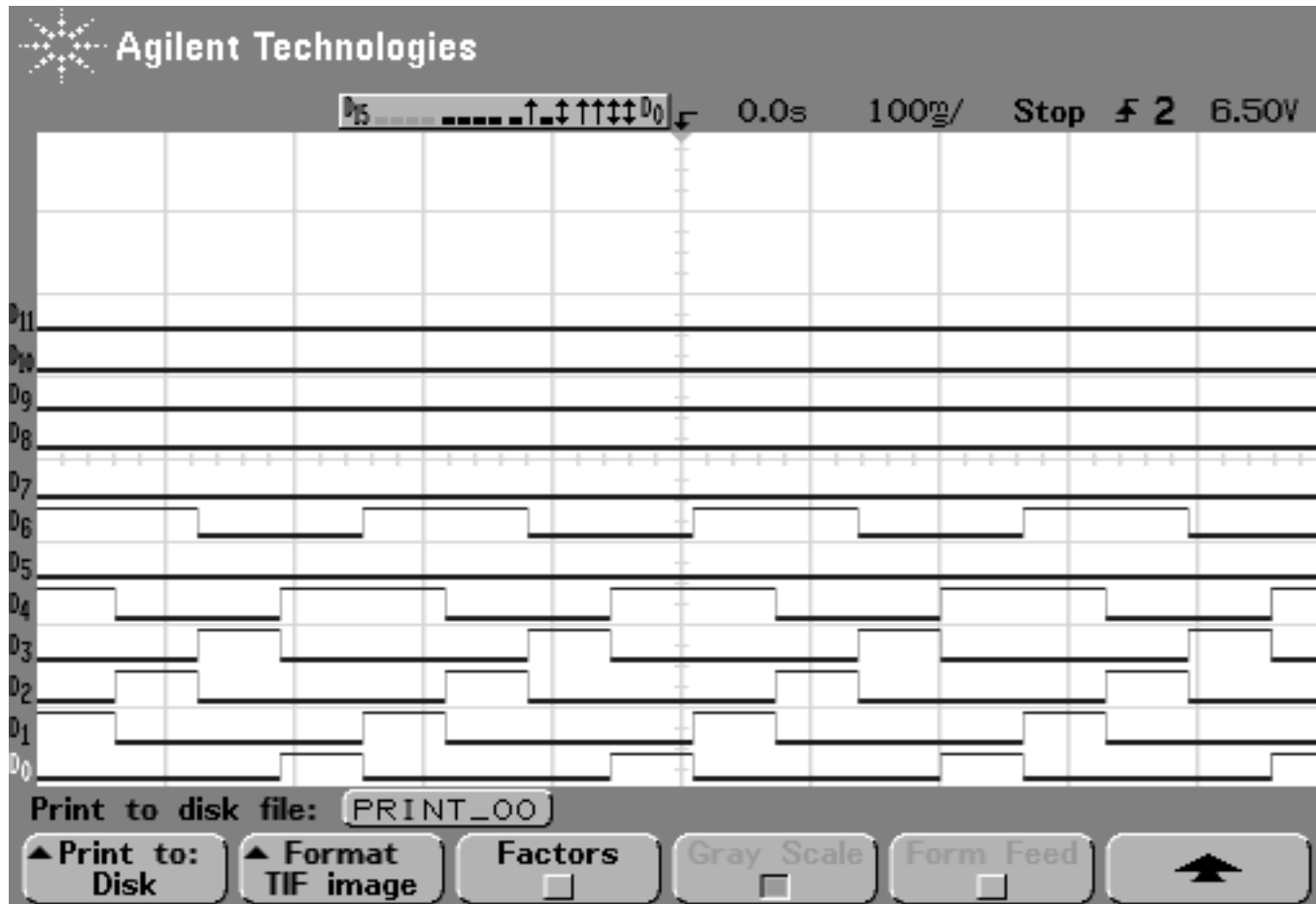
# Case Study Circuit



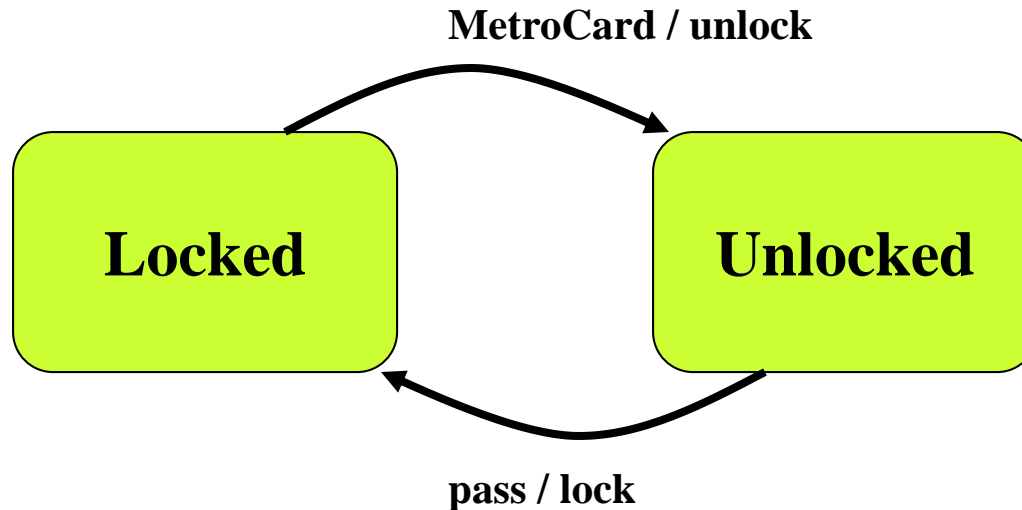
# Oscilloscope Connection



# Oscilloscope Traces

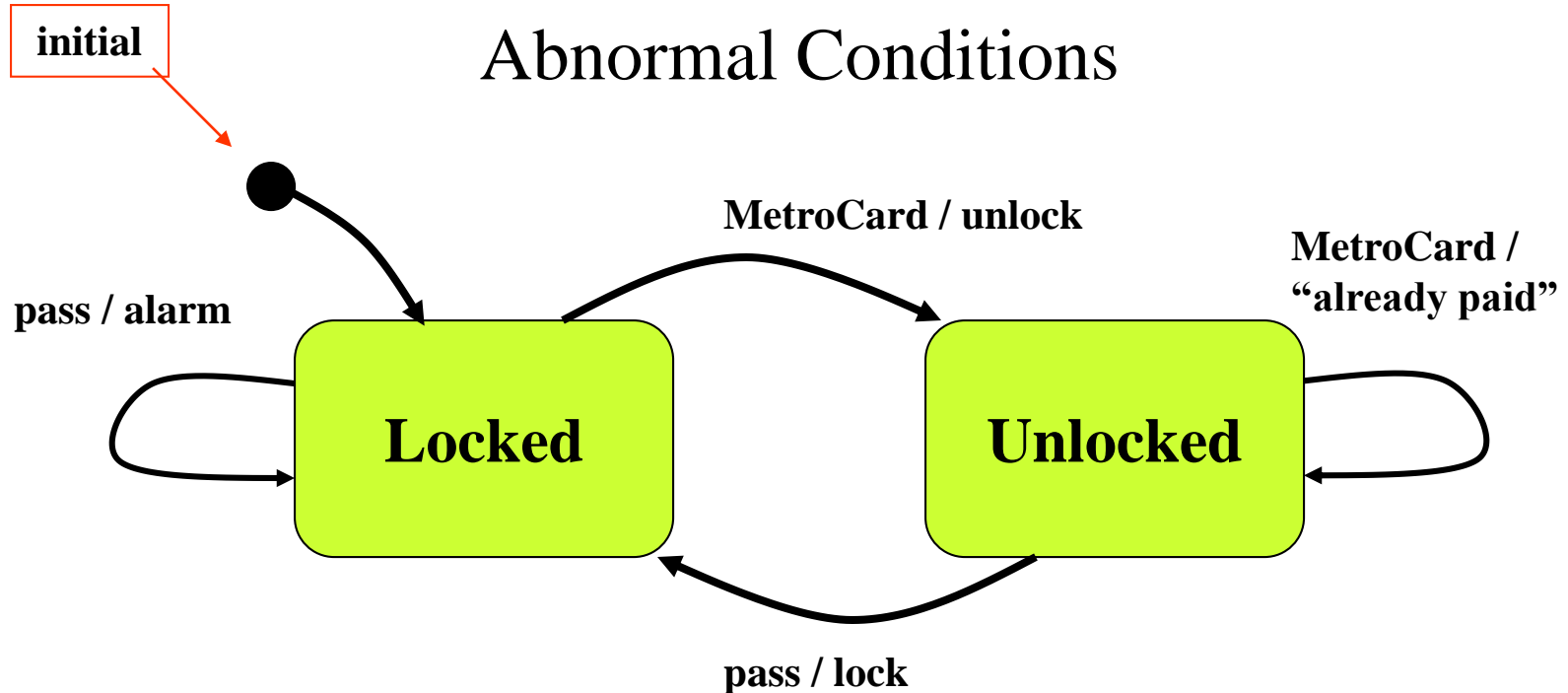


# Finite State Machine Example: Subway Turnstile



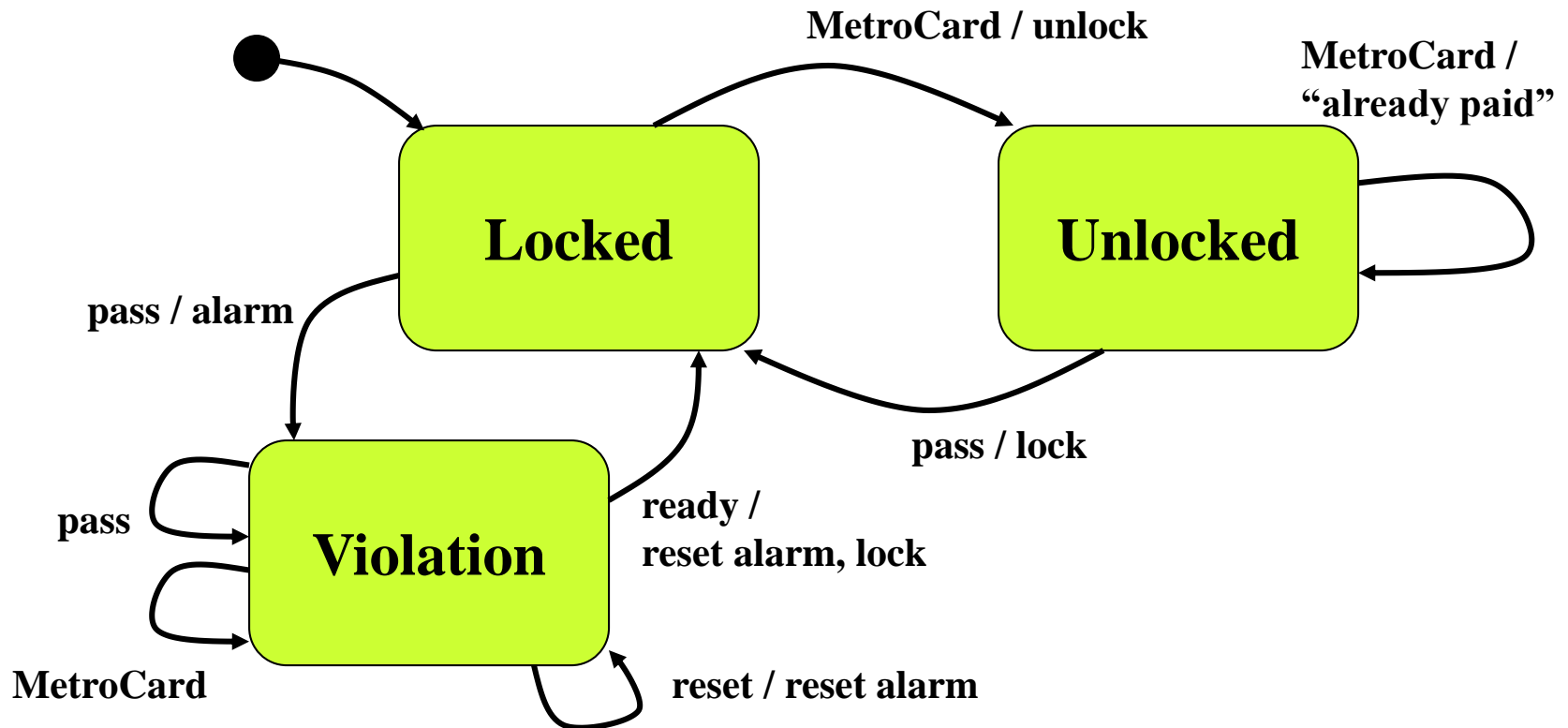
- The label on a transition has two parts separated by a slash. The first is the name of the event that triggers the transition. The second is the name of an action to be performed once the transition has been triggered.
  - If the turnstile is in the Locked state, and a MetroCard event occurs, then the turnstile transitions to the Unlocked state, and the unlock action is performed.
  - If the turnstile is in the Unlocked state, and a Pass event occurs, then the turnstile transitions to the Locked state, and the lock action is performed.

# Abnormal Conditions



- What should we do if the turnstile is in the Locked state, but the user passes through anyway? Sound some kind of alarm? Note that the transition that handles this does not change the state. The turnstile remains in the Locked state.
- The other abnormal condition is when the turnstile is already unlocked and the customer swipes his MetroCard again. In this case, we light up a little “already paid” light.

# Advanced Alarm Conditions

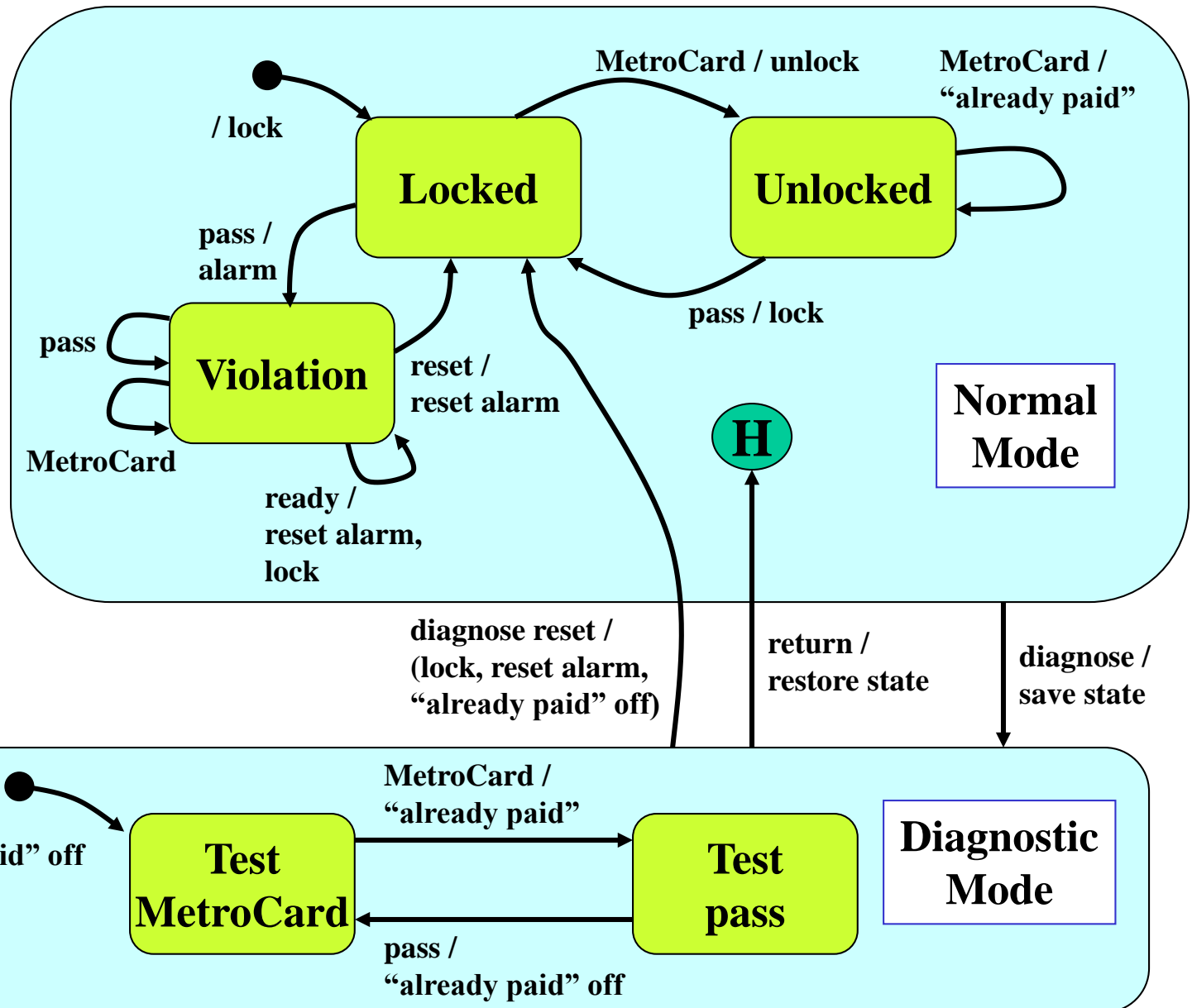


Remaining in the Locked state may not be the best approach for dealing with someone who has forced entry through the turnstile. We could enter some kind of Violation state. We can remain in that state until someone turns off the alarm and signals that the turnstile is ready for service.



# Handling Diagnostics

2 super states



# Example - Diet

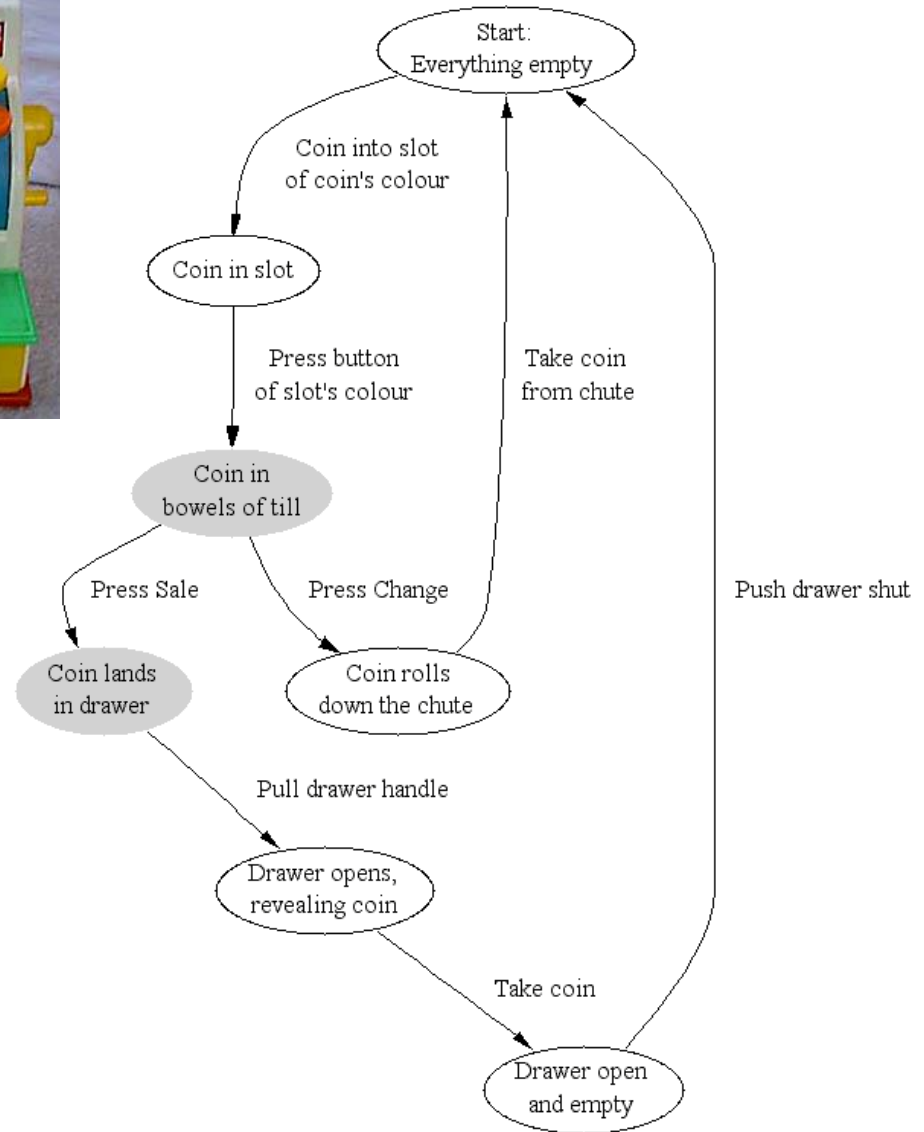


You are on a crazy diet. On any given day you can only eat 1 thing: either cheese, lettuce or grapes. If you go to buy food, you can only buy what you are eating that day. Each day you can decide what you are eating the following day. This is illustrated in this state machine.

# Toy Cash Register



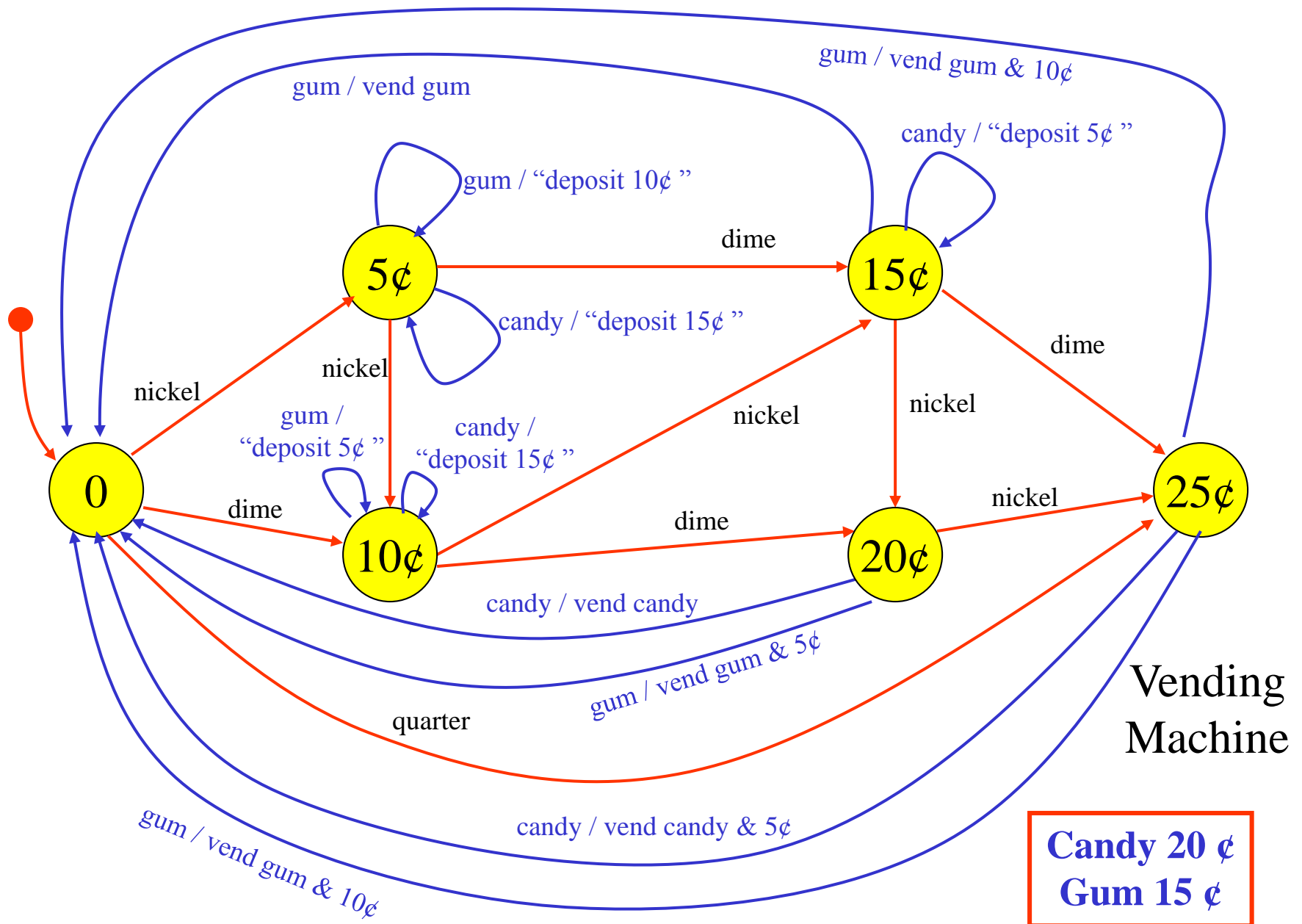
There are 3 slots for coins. Each slot takes a coin of a particular color, and the sizes of the slots and coins constrain you to match colors. When you put the coin in the slot it rests on a little ledge until you press a button just underneath the slot, which makes the coin drop inside the cash till. You can have up to one coin per slot inside the till. When one or more coins are inside you can get them to drop into the drawer by pressing the **Sale** button, or slide down a chute on the side by pressing the **Change** button. To open the drawer you turn a handle on the other side from the chute, and to close the drawer you just push it.



# Vending Machine

- Accepts coins – nickel, dime, quarter
- Vends gum & candy
  - Gum is 15 ¢
  - Candy is 20 ¢
- Gives change

**Operation is difficult to describe with procedural code.  
Naturally lends itself to a State Machine formulation.**



# Describing a system as a state machine

## Vending Machine Example

- 1. List all possible states** {0 ¢, 5 ¢, 10 ¢, etc.}
- 2. Declare all variables** {can have a variable “display” which can take values “add 5 ¢”, “VEND”, “add 10 ¢”, etc. }
- 3. For each state, list possible transitions (with conditions) to other states (define inputs)** {inputs – deposit coins, request products}
- 4. For each state and/or transition, list the associated actions (outputs)** {output - vending gum or candy, or do nothing}
- 5. For each state, ensure exclusive and complete exiting transition conditions**
  - No two exiting conditions can be true at same time
    - Otherwise non-deterministic state machine
  - One condition must be true at any given time
  - Transition on a clock signal is implicit

- A Deterministic Finite State Machine (FSM) is a 4-tuple consisting of:
  - a finite set of states (  $S$  )
  - a finite set called the alphabet (  $\Sigma$  ) – the inputs
  - a transition function  $T$  (  $T: S \otimes \Sigma \rightarrow S$  ) – applying an input to a state transitions to another state
  - a starting state ( $s_0 \in S$  )
- Let  $M$  be a FSM such that  $M = (S, \Sigma, T, s_0)$
- Let  $X = x_0, x_1, \dots, x_n$  be a string over the alphabet  $\Sigma$  (e.g. sequence of inputs)
- $M$  accepts the string  $X$  if the sequence of states  $r_0, r_1, r_2, \dots, r_n$  exists in  $S$ 
  - $r_0 = s_0$
  - $r_{i+1} = T(r_i, x_i)$  for  $i = 0, 1, 2, \dots, n$
- Adding a set of final states  $A$  makes it a 5-tuple
  - a set of accept states (or final states) (  $A \subseteq S$  ) (possibly empty)
  - $r_n \in A$
- Adding outputs  $\Lambda$  makes it a 6-tuple
  - a finite set called the output alphabet ( $\Lambda$ )
  - an output function ( $G: S \otimes \Sigma \rightarrow \Lambda$ )

## Formal Definition

# Mealy & Moore Machines

- Moore machine [Edward F. Moore “Gedanken – Experiments on Sequential Machines” (1956)] (Gedanken = thinking)
  - output only depends on the state
- Mealy machine [G. H. Mealy “A Method for Synthesizing Sequential Circuits” (1955)]
  - output depends on both the input and the state
  - You can often simplify a Moore machine to be a Mealy machine



# Embedded C Programming

# Why Assembly Language?

- Execution Speed
  - Closer to Microcomputer Architecture
  - Compact Code
  - Determinism
  - Fine Manipulation of Data
- 
- It will make you a better embedded programmer.

# Why C Programming?

- ANSI Standard
- Portability
- Programming Efficiency
- Readability
- Optimizing Compilers
- C++ Extensions
- Real Time Operating Systems

# Hi-Tech PICC C Compiler

## Data Types

Type	Size (in bits)	Arithmetic Type
<code>bit</code>	1	boolean
<code>char</code>	8	signed or unsigned integer <sup>a</sup>
<code>unsigned char</code>	8	unsigned integer
<code>short</code>	16	signed integer
<code>unsigned short</code>	16	unsigned integer
<code>int</code>	16	signed integer
<code>unsigned int</code>	16	unsigned integer
<code>long</code>	32	signed integer
<code>unsigned long</code>	32	unsigned integer
<code>float</code>	24	real
<code>double</code>	24 or 32 <sup>b</sup>	real

**You should always avoid using real numbers in a fixed point processor.**

You should always choose the smallest data type that can hold your data.

# Hi-Tech PICC C Compiler

## Radix Formats

Radix	Format	Example
binary	<i>0bnumber</i> or <i>0Bnumber</i>	0b10011010
octal	<i>0number</i>	0763
decimal	<i>number</i>	129
hexadecimal	<i>0xnumber</i> or <i>0Xnumber</i>	0x2F

Configuration bits can be set in a number of ways:

```
--_CONFIG (11111110110010)
```

```
--_CONFIG(WDTDIS & XT & UNPROTECT);
```

# Functions

- The C language is formulated as function calls.
- Format:

```
return_type    FunctionName (parameters)
{
    Statements in Functions are
surrounded by curly brackets.
}
```

- If the function returns a variable to the routine that calls it, “return\_type” is the data type of the variable returned.
- The “parameters” are variables that are passed to the function from the routine that calls it.

# Main Function

- Every C program must have a **main** function. (The compiler will look for it.)
- Format:

```
void    main (void)
{
}
```

- Nothing can call **main** thus there are no parameters.
- **main** cannot return anything (return\_type is void).
- Typically put **main** function last in file so all functions called are already defined to compiler.
- Program execution starts with the first line of **main**.

# Statements & Comments

- Statements end in semi-colon ;
- Comments

Everything between `/*` and `*/` is a comment

Everything between `//` and the end of the line is a comment

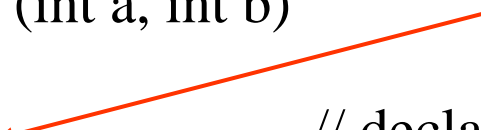
- **# Statement** is a directive to the compiler
  - # Define** is equivalent to EQU in Assembler
  - # Include <filename>** tells the compiler to include another file with the code you write
- C ignores whitespace



# Function Example

```
int Adder (int a, int b)
{
    int p ;           // declare p an integer
    p = a + b ;       // set p equal to the sum of a & b
    return p ;        // return the value p to the calling program
}
```


**integer p is local to function Adder**



```
int a = 5, b = 6, c ;    // declare a, b & c and initialize a & b

void main (void)
{
    c = Adder (a, b) ;    // will set c = 11
}
```

**integers a, b & c are global to program**



# Assignment Operator

Assignment is a form of an expression

Examples

$a = b ;$

$a = 10 ;$

$a = b + 10 ;$

Assignment Operator

**$\text{expression}_1 \text{ operator} = \text{expression}_2$**

is equivalent to

**$\text{expression}_1 = \text{expression}_1 \text{ operator } \text{expression}_2$**

Example

$a * = b + 10 ;$

is equivalent to  $a = a * (b + 10) ;$

NOT  $a = a * b + 10 ;$

# C Functions are “Call by Value”

**As opposed to Fortran &  
Pascal in which functions are  
“Call by Reference”.**

```
int  Increment (int a)
{
    int p ;           // declare p an integer
    a = a + 1 ;       // increment a
    p = a ;           // set p equal to a
    return p ;        // return the value p to the calling program
}
```

```
int  a = 5, c ;       // declare a & c and initialize a
```

```
void  main (void)
{
    c = Increment (a) ; // will set c = 6
}
```

**integer a is still equal to 5**

# Arithmetic Expressions

## Arithmetic operators

- +**    **addition**
- **subtraction**
- \***    **multiplication**
- /**    **division**
- %**    **modulus operator**

$x \% y$   
produces the remainder when  
x is divided by y  
 $20 \% 6 = 2$   
 $9 \% 5 = 4$

Division of integers truncates the result.  
(fractional part is discarded)

$$20 / 6 = 3$$
$$9 / 5 = 1$$

# Increment & Decrement Expressions

Increment and decrement

**++**      **adds 1**  
**--**      **subtracts 1**

Examples

`n++;`                      `// is equivalent to n = n + 1`

postfix

`int n = 5;      // n initialized to 5`  
`x = n++;        // x = 5 & n = 6`

**when you set  
x = right hand side.**

prefix

`int n = 5;      // n initialized to 5`  
`x = ++n;        // x = 6 & n = 6`

# Relational Operators

Result is either **FALSE (0)** or **TRUE (1)**

**>** is greater than

**>=** is greater than or equal to

**<** is less than

**<=** is less than or equal to

**==** is equal to

**!=** is not equal to

**a = 5;**

sets a equal to 5

**a == 5;**

is **TRUE** if a equals 5  
otherwise **FALSE**

## Examples

```
int n = 5, m = 10;    // n initialized to 5, m to 10
n > m;                // is FALSE
n <= m;               // is TRUE
n == m;              // is FALSE
```

lower than arithmetic

**n > lim - 1;**

evaluate first

# Logical Operators

Treat non-zero value as **TRUE** and zero value as **FALSE**

**&&**

**AND**

**||**

**OR**

**!**

**Logical Negation**

Examples

**(10 > 5) && (-1 > -2)**

**// is TRUE**

**(10 < 5) || (-1 > -2) && (2 < 1)**

**// is FALSE**

**! Open**

**// is TRUE if Open is FALSE**

Unusual example

**10 && 20**

**// is TRUE**

# C Logical Expressions

- Logic operators are evaluated using *lazy evaluation*.

Lazy evaluation – Once a value completes the condition, stop

OR        any condition is found to be TRUE

AND      any condition is found to be FALSE

- Why is lazy evaluation important?
  - Makes code run faster – skips unnecessary code
  - Know condition will / will not evaluate, why evaluate other terms
  - Can use lazy evaluation to guard against unwanted conditions



# Variable Sizes

- Smallest unit of information is a bit
  - Base 2 notation  $\Rightarrow$  Two values
  - 1 (TRUE) 0 (FALSE)
- Nibble
  - 4 bits  $\Rightarrow$  16 possible values
  - Ratio  $\Rightarrow$  1 nibble is 1 hexadecimal character
- Byte (character)
  - 8 bits  $\Rightarrow$  256 possible values
- Word (integer)
  - 16 bits  $\Rightarrow$  65,536 possible values (65 K)
- Double Word (long)
  - 32 bits  $\Rightarrow$  4,294,967,296 possible values (4 M)

# Using Bitwise Operations

- Bitwise Operators
  - Bit manipulation (as is done in Assembler) is a key component of embedded programming.
- Efficient Use of Memory
  - Instead of using 8 bits to store one value, now we can use individual bits to store information.
- Operations are done on a bit by bit basis.
  - Hence the name bitwise operators / manipulation
- Binary or Hex notation is commonly used.
  - 0xFF is 11111111 in binary
  - 0x10 is 00010000 in binary
  - Recall the binary → hex conversion (hex is 4 binary digits)

# Bitwise Operators

Operation is done on all bits in the variable

**&**     **AND**

**|**     **OR**

**Easier to see in Binary**

**^**     **EXCLUSIVE OR (XOR)**

**~**     **Ones Complement (Negation)**

**<<**     **Left Shift (pad with zeroes)**

**>>**     **Right Shift (pad with zeroes)**

Examples

**0x10 & 0x10 = 0x10**

**0x01 | 0x10 = 0x11**

**0xF0 ^ 0x10 = 0xE0**

Examples

**00010000 & 00010000 = 00010000**

**00000001 | 00010000 = 00010001**

**11110000 ^ 00010000 = 11100000**

# Bitwise Operations Notes

- AND Operator  $\&$ 
  - 0 ANDed with anything will always give a 0
  - 1 ANDed with anything will give the same value as the original
- OR Operator  $|$  (pipe symbol)
  - 1 ORed with anything will always give a one
- EXCLUSIVE OR Operator  $\wedge$ 
  - 1 XORed with anything will toggle the bit

# Bank Switching

- C Compiler takes care of switching between banks.
- Special Function Registers still have to be set up for proper operation
- C Compiler takes care of assigning names to registers
- In Simulator, you will want to open Watch window with name.

## Register Examples

```
TRISB    = 0B00001111 ;  
PORTB    = 0B11111111 ;  
TRISC    = 0B00000000 ;  
PORTC    = 0B01010101 ;
```

## Bit Examples

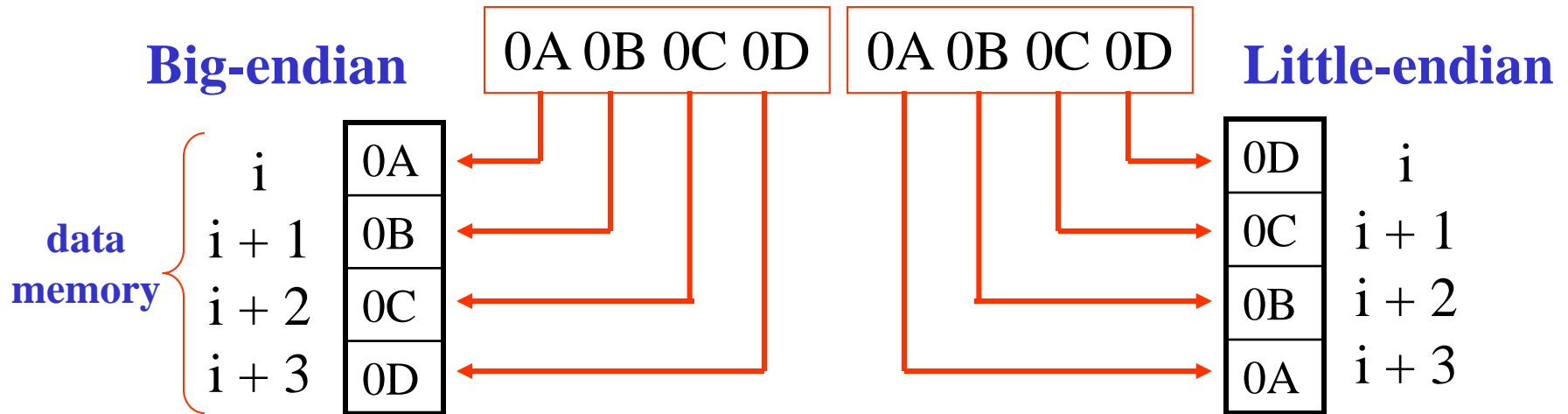
```
RB0      = 0 ;  
ADGO     = 1 ;
```

## Assignment Examples

```
char Temp, Count;  
int i, Num;
```

# Data Memory Layout

- Byte-ordering: Big-endian vs. Little-endian
  - Big-endian: Byte 0 of a n-byte variable is the Most Significant Byte (MSB)
  - Little-endian: Byte 0 of a n-byte variable is the Least Significant Byte (LSB)



- Byte is still ordered left to right Most Significant Bit (MSb) to Least Significant Bit (LSb)



# Data Memory Layout

- **Alignment:** A scalar variable of  $n$  bytes must be aligned on  $n$ -byte boundaries

Examples:

char: no restriction (any byte register in memory)

int or short: two-byte boundaries

long: four-byte boundaries

float: three-byte boundaries

**HiTech Compiler is Big-endian**

# Operator Precedence

~	!	- (unary)	++	--	
*	/	%			} arithmetic
+	-				
<<	>>				→ bit shift
<	<=	>	>=		} relational
==	!=				
&					} bitwise logical
^					
&&					} Boolean



# Bitwise Shifting Operations

- Note: The shift operation may be done via an arithmetic shift or by a logical shift (both are ANSI standard)
  - Arithmetic – MSb stays the same on a right shift
  - Logical – Always shift in a zero
- Arithmetic shift is done if variable is declared as signed

## Examples

```
0B00001111 >> 2 = 0B0000011 ;  
0B00001111 << 2 = 0B00111100 ;  
0B10011111 << 2 = 0B01111100 ;
```

## Examples

```
signed char Count = - 12;  
Count = 0B11110100  
Count >> 2 = 0B11111101 ( - 3 )  
Count << 2 = 0B11010000 ( - 48 )
```

# Control Flow

- Simple Statements end in a semi-colon ;
- Compound statement (Curly brackets group statements)

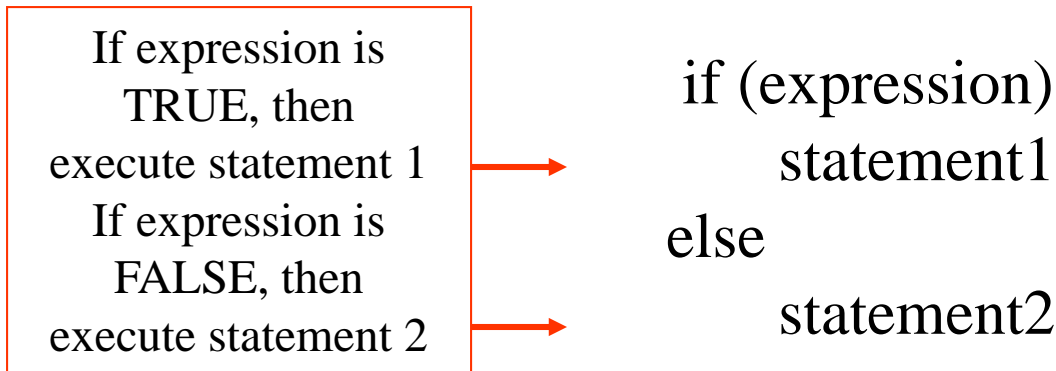
```
{  
    declarations;  
    statements;  
}
```

## Example

```
if (n > 0)  
{  
    int Count = 0, i;  
    Count++;  
    i = Count;  
    ...  
}
```

# If – Else → Control Flow

- If-else statement



## Example

```
if (n > 0)
{
    if (a > b)
        z = a;
    else
        z = b;
}
```

# Else – If → Control Flow

- Else-if style

```
if (expression1)  
    statement1  
else if (expression2)  
    statement2  
else if (expression3)  
    statement3  
else  
    statement4
```

# Switch → Control Flow

- Switch statement

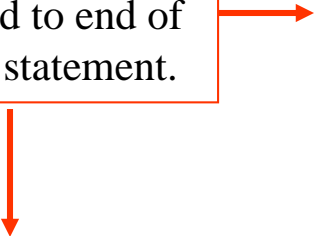
Once one  
constant-expression  
is TRUE, then  
ALL statements are  
executed to end of  
Switch statement.

```
switch (expression)  
{  
    case constant-expression: statements  
    case constant-expression: statements  
    ...  
    default: statements  
}
```

# Switch → Example

```
switch (count)
{
    case 1: statements
    case 2: statements
    case 3:
        {
            statements
        }
    case 4: statements
    ...
    default: statements
}
```

Once one  
constant-expression  
is TRUE, then  
ALL statements are  
executed to end of  
Switch statement.



# While → Control Flow

- While loop

**while** (*expression*)  
*statement*

Example

```
i = 0;  
Count = 0;  
while (i < N)  
{  
    if (a > b)  
        Count ++;  
    i ++;  
}
```

Example

```
while (RedButton == 1) { }
```


Example

```
while (!RedButton) { }
```

# For → Control Flow

- For loop

```
for (expression1; expression2; expression3)  
    statement
```



If you leave out **expression1** or **expression2**, you must put semi-colons to mark place.

- Equivalent

```
expr1;  
while (expr2)  
{  
    statement  
    expr3;  
}
```



# Do – While → Control Flow

- Do-while loop

**do**

**statement**

**while (expression) ;**

**The main difference between the do and the while loops is that in the do loop, the statement is executed whether or not the while condition is TRUE or FALSE.**

Example

```
i = 0;  
Count = 0;  
do  
{  
    if (a > b) i ++;  
    Count ++;  
}  
while (i < N) ;
```

# Fine Control Flow

- **Break**: Exit the innermost loop or switch
- **Continue**: Start the next iteration immediately

**Recall that in a switch statement, all code after a constant-expression is TRUE is executed. Break allows exit from the switch.**

# Goto → Control Flow

- Goto and labels
- You can always write code without using goto.
- Most C textbooks say that they should be only used rarely!
- I do not care.
- Labels end in a colon :

Example

Mode0:

Count = 0;

if (GreenButton)

goto Mode1;

Count ++;

.....

Mode1:

.....

# Conditions and Testing

- Comparison – Multiple Conditions
- Tie together using Boolean (logical) operators
  - `&&`      AND
  - `||`        OR
  - `!`         NOT

## Example

```
if ((nVal > 0) && (nArea < 10))
```

```
if ((nVal < 3) || (nVal > 50))
```

```
if ( ! (nVal <= 10))
```

# Variable Scope and Storage

- **scope:** Where a variable is visible
- Global data:
  - Declared outside all C functions
  - Visible to all C functions
- Variables with static storage
  - Visible to functions in same file
- Local Variables
  - Defined within a function
  - Visible only within function.

# Memory Pointers

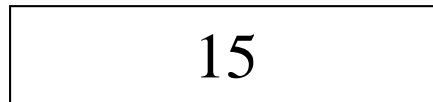
- How to use pointers

\*Temp = 15

&Temp = 20

20

Temp



Temp EQU 20