Embedded Microprocessor Systems

Module: EE2A2 Embedded Microprocessor Systems

Lecturer: James Grimbleby
URL: http://www.personal.rdg.ac.uk/~stsgrimb/
email: j.b.grimbleby@reading.ac.uk

Number of Lectures: 10

Recommended text book:
The Design of Small-Scale Embedded Systems
Tim Wilmshurst
Palgrave 2001
Embedded Microprocessor Systems

The Design of Small-Scale Embedded Systems
Tim Wilmhurst
Palgrave 2001

Approx. price: £35
Microprocessor Syllabus

This course of lectures deals with the design of systems containing embedded microprocessors or microcontrollers.

The topics that will be covered include:
- Microprocessor bus systems and memory maps
- Memory usage and technologies
- Address decoding
- The stack, subroutines and stack frames, the heap
- Exceptions and exception processing
- Interrupts
- Serial interfacing
- Number representations
- Product development cycle
- Microprocessor programming
- Safety-critical systems
Microprocessor Prerequisites

You should be familiar with the following topics:

**SE1EB5: Computer and Internet Technologies**
- Binary codes
- Boolean algebra
- Karnaugh maps and Boolean simplification
- Logic gates
- Flip-flops

**SE1EC5: Engineering Mathematics**

**SE1SA5: Programming**
- Basic programming in C
**Hexadecimal Notation**

<table>
<thead>
<tr>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
<th>Dec</th>
<th>Bin</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
<td>10</td>
<td>1010</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
<td>11</td>
<td>1011</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
<td>12</td>
<td>1100</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
<td>13</td>
<td>1101</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
<td>14</td>
<td>1110</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
<td>15</td>
<td>1111</td>
<td>F</td>
</tr>
</tbody>
</table>

Decimal address: 12853072  
Binary: 1100 0100 0001 1111 0101 0000  
Hex: C 4 1 F 5 0  
C++ notation: 0xC41F50
Microprocessors

Microprocessors are processors integrated onto a single semiconductor chip.

Examples: Freescale Coldfire, Intel Pentium, ARM

Microprocessors normally have no memory or peripheral interfaces on-chip.

Systems based on microprocessors are more complex and expensive than systems based on microcontrollers.

However, they are more flexible because the designer can select the memory sizes and peripherals.
Microcontrollers

Microcontrollers are processors, memory and peripheral interfaces integrated onto a single semiconductor chip.

Examples: Microchip PIC, Intel 8051

Systems based on microcontrollers are normally simpler and cheaper than systems based on microprocessors.

Memory sizes and interfaces are fixed by the manufacturer.

This means that manufacturers have to provide many variants to meet most requirements.
Bus Systems

Microprocessors and microcontrollers communicate with memory and peripherals using a bus system.

Microcontrollers provide no access to the internal address/data buses: memory and I/O is all on-chip.

Microprocessors require memory and peripherals to be connected via an external bus system.

A microprocessor bus system will normally consist of three elements: address bus, data bus and control bus.
Microprocessor Bus System

- **Data Bus**
- **Address Bus**
- **Control Bus**

- Microprocessor
- Non-volatile memory
- Read-write memory
- Interface
  - External Device
- Interface
  - External Device
Data Bus

The data bus is used for transferring data between microprocessor and external devices such as memory.

Most embedded microprocessors have either an 8-bit or 16-bit data bus.

The data bus is bi-directional and data can flow either from microprocessor to devices or devices to microprocessor.

Devices which can be read by the processor must have tri-state logic connecting to the data bus to avoid contention.
Address Bus

The address bus is used to specify the device, and the location within the device, that is being accessed.

Most embedded microprocessors have either a 16-bit, 24-bit, or 32-bit address bus.

The address bus is unidirectional and is controlled by the microprocessor.

All devices must monitor the address bus, and only respond when the address assigned to them appears on the bus.
A typical control bus might have the following lines:

**RESET**: (active-low) puts the processor and devices in a well-defined initial state on power-up

**R/W**: read not write - defines the direction of data flow

**DAV**: (active low) data valid - data on the data-bus is valid

**DTA**: (active low) data transfer acknowledge - data transfer is complete

**IRQ**: (active low) interrupt request
Synchronous Bus

Simple 8-bit microprocessors normally use a synchronous bus system with bus transfers timed by the microprocessor clock.

The Motorola MC6800 and MC6809 microprocessors are examples of devices using a synchronous bus.

In synchronous buses a DAV line of the control bus tells peripherals that the information on the address, data and R/W lines is valid.

The DAV line is held active by the microprocessor for a fixed time, and bus devices must respond within this time.
Synchronous Bus

Microprocessor read cycles:

- **A-bus**: Address invalid
- **R/W**: Read/Write
- **DAV**: Data available
- **D-bus**: Data invalid
Synchronous Bus

Microprocessor write cycles:

A-bus

R/W

DAV

D-bus
Synchronous Bus

The primary advantage of synchronous bus operation is its simplicity.

The maximum data transfer rate is determined by the slowest device.

A typical synchronous bus operates on a 250 ns cycle time and bus devices have less than 125 ns to respond.

Some microprocessors allow "wait states" to be inserted during bus transfers to slow devices.
Asynchronous Bus

In asynchronous microprocessor buses the data transfers occur at a rate determined by the communicating devices.

Asynchronous buses need an extra control line DTA (data transfer acknowledge) to accomplish variable rate transfers.

DAV is asserted by the microprocessor to inform the devices that the address (and possibly data) is valid.

DTA is then asserted by a device to inform the microprocessor that the data transfer is complete.

DTA is active low and driven by open-collector (wired-OR) logic gates.
Asynchronous Bus

Microprocessor read cycles:

A-bus

R/W

DAV

DTA

D-bus
Asynchronous Bus

Microprocessor write cycles:

A-bus
R/W
DAV
DTA
D-bus
Microprocessor Memory

Microprocessor memory is normally byte-addressable, even if the processor has a 16 or 32-bit data bus.

Accessing a 16-bit word or a 32-bit long word involves 2 or 4 memory locations respectively.

<table>
<thead>
<tr>
<th>Address</th>
<th>Big endian</th>
<th>Little endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200003</td>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x200002</td>
<td>0xFF</td>
<td>0xC7</td>
</tr>
<tr>
<td>0x200001</td>
<td>0xA0</td>
<td>0xC7A0</td>
</tr>
<tr>
<td>0x200000</td>
<td>0xC7</td>
<td>0xC7A0FF00</td>
</tr>
</tbody>
</table>
Memory Maps

All devices in microprocessor or microcontroller systems are memory-mapped

Each device responds to a particular memory address or block of addresses

A read-write memory chip of capacity $2^{16}$ bytes might be mapped to address block $0x100000$ to $0x10FFFF$

Memory maps for microcontrollers are normally fixed by the manufacturer

Memory maps for microprocessor systems are created by the system designer
Microprocessor System Memory Maps

A microprocessor system has its program stored in non-volatile memory

Non-volatile memory is also be used to store the exception vectors (at fixed locations starting from address 0x000000)

Non-volatile memory is therefore normally mapped to the lowest memory addresses

Read-write memory is used to store program variables, data stack and heap

All other devices connected to the bus (including read-write memory) can be mapped to any convenient memory blocks
Microprocessor System Memory Maps

Keypad (2 byte)
Stepping motor (1 byte)
7-segment display (8 byte)
8-bit DAC (1 byte)
Read-write memory (16 kB)
Non-volatile memory (128 kB)
Some microcontrollers use Harvard architecture: they have separate program and data/interface memory spaces.

The non-volatile memory is mapped to the program space and contains the program and exception vectors.

The read-write memory and peripheral interfaces are mapped to the data area.
Microcontroller Memory Maps

Program Space
- 0x17FFF
- 0x01FF
- 0x00200
- 0x00000

Data Space
- 0x7FFF
- 0x0800
- 0x07FF
- 0x0000

Special function registers

Read-write memory

Interrupt vectors
Properties of the ideal microprocessor memory:

1. Fast: read and write cycles take a negligible time
2. Dense: large memory capacity in a small volume
3. Non-volatile: data is retained when power is removed
4. Low power consumption
5. Reliable
6. Inexpensive
Microprocessor System Memory

Semiconductor Memory

Volatile Memory
- Static RAM
- Dynamic RAM

Non-Volatile Memory
- ROM
- EPROM
- Flash
- PROM
- EEPROM

RAM: random-access (read-write) memory
ROM: read-only memory
PROM: programmable ROM
EPROM: erasable PROM
EEPROM: electrically-erasable PROM
The Memory Array

- **Row address** (p bits)
- **Data interface**
- **Data**
- **Column address** (q bits)
- **Word line**
- **Bit line**
- **Memory cell** (1 bit)
- **Corresponds to 1 unique address**

Diagram showing the memory array with components such as row decoder, column decoder, and data interface.
Static Random-Access Memory

Each bit of data is stored in a separate transistor flip-flop consisting of 6 MOSFETs:
Static Random-Access Memory

Data is retained as long as power is supplied to the device, but is lost when power is removed.

A small re-chargeable battery can be used to maintain power to the RAM when the rest of the system is powered down.

Most static RAMs are byte-organised, that is they have 8 bits of data stored at each memory address.

A typical RAM is the TC551001 which has a capacity of 128 Kbyte.

Most embedded systems use static, rather than dynamic memory.
Static RAM: Toshiba TC551001

1 Mbit static RAM
Organised as 128K words of 8 bits (bytes)
Access time: 70 ns
Dynamic Random-Access Memory

Each bit of data is stored in a capacitor

Dynamic RAM is denser and cheaper per bit than static RAM, but:

Charge stored on capacitor leaks away

Data must be refreshed at regular intervals

DRAM uses 1 MOSFET per bit
Programmable Read-Only Memory

Originally ROMs were mask-programmed by the manufacturer in accordance with data supplied by the user.

Programmable ROMs (PROMs) can be programmed by the user.

Each data bit is determined by the state of a fusible link.

Fuses are blown in a special-purpose programmer.

PROM uses 1 MOSFET per bit.
Erasable Programmable Read-Only Memory

EPROM makes use of a floating-gate MOSFET:

![Diagram of floating-gate MOSFET]

- Control gate
- Drain
- Gate
- Source
- Floating gate
- Insulator \( \text{SiO}_2 \)
- Substrate

Erase using short-wavelength ultra-violet light: photoelectric emission gives floating gate a +ve charge

Program by breaking down the drain-substrate diode: hot electrons cross insulator giving floating gate a -ve charge
Erasable Programmable Read-Only Memory

EPROM makes use of a floating-gate MOSFET:

If floating gate is programmed (-ve) the MOSFET cannot be turned on.

If floating gate is erased (+ve) the MOSFET operates normally.

If floating gate is programmed then the bit line is pulled low when word line is high.

EPROM uses 1 MOSFET per bit.
1Mbit EPROM
Organised as 128 Kwords of 8 bits (bytes)
Access time: 90 ns
Flash Memory

Flash memory uses asymmetric floating-gate MOSFETs

Erase is by Fowler-Nordheim tunnelling which establishes a positive charge on the floating gate
Flash Memory

Flash memory is normally erased as a whole, but some devices are split into sectors which can be individually erased.

Flash memory can only be reliably re-programmed a limited number of times (typically 10000 erase/write cycles).

Flash memory is replacing EPROM because of the simpler (and therefore cheaper) packaging.

Re-programming flash memory in situ is not completely straightforward:

Flash memory uses 1 MOSFET per bit.
1Mbit Flash
Organised as 128 Kwords of 8 bits (bytes)
Access time: 90 ns
Electrically-Erasable Programmable Read-Only Memory

EEPROM is similar to flash memory and uses floating-gate MOSFETs.

Unlike flash memory the contents of individual memory elements can be erased and re-written.

Each memory element can only be reliably re-programmed a limited number of times (typically 10000 erase/write cycles).

EEPROM requires 2 MOSFETs per bit and is therefore more expensive and bulky than flash memory.
Serial EEPROM is often used to store small amounts of data or system parameters.

4 Kbit EEPROM
Organised as 512 words of 8 bits (bytes)
Interfaced using the 2-wire I²C bus
Access time (random read): 100 μs
Access time (sequential read): 25 μs
Address Decoding

Each device connected to the microprocessor bus occupies 1 or more addresses.

Address inputs to the device are connected to lower-order lines of the address bus.

Each device has a chip select input (CS) which is asserted to select that particular device.

Address decoder drives CS to make device respond to required block of addresses.

Inputs to address decoder are higher-order address lines and DAV.
Address Decoding

The 74HCT138 can be used as an address decoder (assuming 24-bit address bus):

The higher-order address lines are connected to A, B and C, with the most significant line connected to C.

The Y outputs then act as 8 individual chip selects, each spanning 1/8 of the complete memory space.
Address Decoding

74HCT138 truth table, where: $E = G1 \cdot \overline{G2A} \cdot \overline{G2B}$

<table>
<thead>
<tr>
<th>$E$</th>
<th>$C$</th>
<th>$B$</th>
<th>$A$</th>
<th>$Y0$</th>
<th>$Y1$</th>
<th>$Y2$</th>
<th>$Y3$</th>
<th>$Y4$</th>
<th>$Y5$</th>
<th>$Y6$</th>
<th>$Y7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
CS is derived from Y5: asserted (low) when C=1, B=0, A=1, and DAV=0

Thus: A23=1, A22=0, A21=1, corresponding to addresses:

A23  A0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 (0xA00000)
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 (0xA00001)
.....................0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 (0xBFFFFF)
1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 (0xBFFFFF)

This is a block of 2097152 (2M) addresses which is 1/8 of the total address space: 2^{24} = 16M
Address Decoding

Address decoding using HCT138 and 24-bit address bus:

<table>
<thead>
<tr>
<th>CBA</th>
<th>Address block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y0</td>
<td>000 0x000000 → 0x1FFFFFF</td>
</tr>
<tr>
<td>Y1</td>
<td>001 0x200000 → 0x3FFFFFF</td>
</tr>
<tr>
<td>Y2</td>
<td>010 0x400000 → 0x5FFFFFF</td>
</tr>
<tr>
<td>Y3</td>
<td>011 0x600000 → 0x7FFFFFF</td>
</tr>
<tr>
<td>Y4</td>
<td>100 0x800000 → 0x9FFFFFF</td>
</tr>
<tr>
<td>Y5</td>
<td>101 0xA00000 → 0xBFFFFFF</td>
</tr>
<tr>
<td>Y6</td>
<td>110 0xC00000 → 0xDFFFFFF</td>
</tr>
<tr>
<td>Y7</td>
<td>111 0xE00000 → 0xFFFFFFFF</td>
</tr>
</tbody>
</table>

Note: this method of decoding is somewhat inflexible because address blocks must be of equal size.

James Grimbleby  School of Systems Engineering - Electronic Engineering  Slide 46
Address Decoding

A 27C010 EPROM is required to respond to addresses 0xC00000 to 0xC1FFFF:

<table>
<thead>
<tr>
<th>A23</th>
<th>A16</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11000000000000000000000000000000</td>
<td>(0xC00000)</td>
<td></td>
</tr>
<tr>
<td>11000000000000000000000000000001</td>
<td>(0xC00001)</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11000000001111111111111111111111</td>
<td>(0xC1FFFF)</td>
<td></td>
</tr>
</tbody>
</table>

A17 to A23 remain constant over this address range; all the other address lines change.

CS is asserted when A23=1, A22=1, A21=0, A20=0, A19=0, A18=0, A17=0 and DAV=0
Partial Address Decoding

A 27C010 EPROM is required to respond to addresses 0xC00000 to 0xC1FFFF

Address space 0xC20000 to 0xCFFFFF is probably unused: make CS respond to this extended address range

\[
\begin{array}{cccccc}
A23 & A19 & & & & A0 \\
11000000000000000000000000000000000 & (0xC00000) \\
11000000000000000000000000000000001 & (0xC00001) \\
& & & & & (0xC000000) \\
110001111111111111111111111111111 & (0xC000011)
\end{array}
\]

CS is asserted when A23=1, A22=1, A21=0, A20=0 and DAV=0
Partial Address Decoding

EPROM will respond to addresses 0xC00000 to 0xCFFFFFFF
Primary address block: 0xC00000 to 0xC1FFFFF
Mirror address blocks: 0xC20000 to 0xC3FFFFF
0xC40000 to 0xC5FFFFF etc
Address Decoding for an Asynchronous Bus

An extra signal, $DTA$, must be generated to signal to the microprocessor that the data transfer is complete.

Some devices generate $DTA$ themselves, but this is unusual.

$DTA$ must be asserted after the data transfer has taken place, and must therefore be delayed relative to $CS$.

This can conveniently be achieved by the use of a shift register clocked by the system clock.

$DTA$ must be driven by an open-collector gate.
Address Decoding for an Asynchronous Bus

[Diagram showing a circuit with components labeled and connections for address decoding.]
Address Decoding Using GALs

An alternative approach to using discrete logic for address decoders is to use some type of programmable logic device.

This has the advantages of a smaller chip count and allowing changes to CS functions without PCB modification.

In practice a GAL (generic array logic) device is normally used.

A typical GAL, the GAL16V8, has 10 inputs, and 8 outputs.

Outputs can be direct or registered, and have programmable tri-state capability.
GALS are programmed in a special-purpose programmer from data supplied as a JEDEC format file.

The JEDEC file is generated by compiling a high-level definition (HDL) file.
Address Decoding Using GALs

HDL specification for address decoder (CUPL):

/* input pins */
P1 N 1 = clock;
P1 N 2 = a20;
P1 N 3 = a21;
P1 N 4 = a22;
P1 N 5 = a23;
P1 N 6 = dav;

/* output pins */
P1 N 19 = cs;
P1 N 18 = dta;

/* feedback pins */
P1 N 13 = u0;
P1 N 14 = u1;

/* equations */
!cs. d = !dav & a23 & a22 & !a21 & !a20;
u0. d = !cs;
u1. d = u0;
dta = 'b' 0;
dta. oe = u1;
Address Decoding Using GALs
Simple Input/Output Interface

James Grimbleby

School of Systems Engineering - Electronic Engineering
Memory Usage

Fixed:
Used for program code and constants

Static:
Allocated for the entire program run time
Used for global variables

Automatic (stack):
Allocated while code block executes
Used for return addresses in function calls and exceptions
Also used for local variables in functions

Dynamic (heap):
Allocated under program control
The Stack

Logical stack:

- **empty**
- push A
- push B
- push C
- push D

- pull D
- pull C
- push E
- pull E
- pull B
The Stack

Actual stack is implemented in read-write memory by the use of a stack pointer SP:

- **empty**
- **push A**
- **push B**
- **push C**
- **pull C**
The Stack

Most microprocessors have instructions specifically for maintaining a stack.

PIC24 has a `MOV` instruction with register indirect mode and pre-decrement or post-increment:

Push data in W0 onto stack: `MOV W0, [W15++]`
Pull data from stack into W0: `MOV [--W15], W0`

Any one of the 16 general purpose registers W0-W15 can be used as a stack pointer.

However, instructions that implicitly involve the stack such as `RCALL` and `RETFIE` use W15.
The Stack

The stack is used to store the return address in function calls:

Return address \( f_A \)  

Return address \( f_B \)  

empty  

RCALL \( f_A \)  

RCALL \( f_B \)  

RETURN  

RETURN
Subroutines and the Stack

The C++ code below shows a simple function `delay()` with no return value and no parameters:

```cpp
int i;
void delay() {
    i = 0;
    while (++i < 100);
}

int main() {
    delay();
    return 0;
}
```

Note that loop variable `i` is defined globally (not good practice)
Subroutines and the Stack

Compiled code (Microchip C30 compiler):

```c
int i;

void delay() {
    i = 0;
    while (++i < 100);
    sub.w W0, W1, [W15]
    bra les, 0x000286
    mov.w #0x64, W0
    mov.w W0, i
}

int main() {
    delay();
    ll 0 000280
    return 0
}
```

Global variable `i` (static storage)
Subroutines and the Stack

Compiled code - local variables:

```c
void delay() {
    int i;
    i = 0;
    while (++i < 100);
}
```

```c
int main() {
    delay();
    return 0;
}
```

Local variable `i` (register storage)
Subroutines and the Stack

Compiled code - function parameters:

```c
void delay(int d) {
  int i;
  i = 0;
  while (++i < d);
}
```

Formal parameter `d`:

```assembly
00280  EB0080  clr.w W1
00282  E80081  inc.w W1, W1
00284  508F80  sub.w W1, W0, [W15]
00286  35FFFD  bra lts, 0x000282
}
```

Copy actual parameter to W0

```assembly
00288  060000  return
```

```assembly
0028A  200640  mov.w #0x64, W0
0028C  07FFF9  rcall 0x000280
```

```assembly
00292  050000  retlw #0x0, W0
```
Stack Frames

General-purpose registers can be used to pass parameters to functions and for variables local to the function.

If there are insufficient registers for this purpose, a stack frame must be created.

This is an area on the stack reserved for parameters and temporary storage.

PIC24 uses register W14 as a frame pointer.

Instructions LNK and ULNK are for creating and destroying stack frames.
Stack Frames: LNK

Stack pointer (SP): W15
Frame pointer (FP): W14

The instruction \texttt{LNK} creates a stack frame

\texttt{LNK framesize:}

1. Store old value of frame pointer on the stack
2. Copy current stack pointer to frame pointer
3. Add \texttt{framesize} to stack pointer
Stack Frames: ULNK

The instruction **ULNK** destroys a stack frame

**ULNK**:

1. Copy current frame pointer to stack pointer
2. Recover old value of frame pointer from the stack

The area of stack reserved for the stack frame has now been released

The frame pointer has been restored to its original value
Stack Frames

SP ← RA ← RA ← RA ← RA  
   k  k  k  a  a  b  b  b  c  c  c  d  d  
FP  FP  FP  FP  

RETURN

SP ← FP ← FP  
   a  a  b  b  c  c  d  
FP  FP  

ULNK

Sp ←
Stack Frame Access

Stack pointer (SP): W15
Frame pointer (FP): W14

MOV W0, [W14+6]
MOV [W14+2], W0
Subroutines and the Stack

Compiled code - return values:

```c
int k;
int times2(int p) {
    return 2 * p;
}
int main() {
    k = times2(20);
    return 0;
}
```

- **Return value in W0**: The compiled code returns the result of `times2(20)` in W0.
- **Copy return value W0 to k**: The result is then copied from W0 to the variable `k` in the `main()` function.
Subroutines and the Stack

Compiled code - reference parameters:

```c
int k;
void times2(int *p) {
    *p *= 2;
}

int main() {
    k = 20;
    times2(&k);
    return 0;
}
```

Copy address of actual parameter \( k \) to W0

Copy address of formal parameter \( p \) to W1
Recursive Functions

Recursive functions (or subroutines) are functions which can call themselves:

```c
int gcd(int a, int b)
{
    int c;
    c = a % b;
    if (c == 0) return b;
    else return gcd(b, c);
}
```

If the local variables are stored in fixed locations then their values will be corrupted when the subroutine calls itself.

If the variables are stored in a stack frame then a new set of local variables will be created for each level of recursion.
Stack Overflow

An area of read/write memory is set aside for the stack.

If this area is insufficient the stack pointer may encroach on areas of memory reserved for global variables, or may go outside the read/write memory area altogether.

As a result data and return addresses will be corrupted.

To prevent this it is necessary to estimate carefully the maximum amount of stack required (and then double it!)

Recursive functions are particularly prone to causing stack overflow and should be converted to non-recursive form.
Recursion Removal by Compiler

```c
int gcd(int a, int b) {
    int c;
    c = a % b;
    if (c == 0) return b;
    else return gcd(b, c);
}
```

Loop:
no RCALL
no recursion

James Grimbleby  School of Systems Engineering - Electronic Engineering  Slide 78
The Heap

The heap is an area of read/write memory used for unstructured data storage

In C++ heap memory is allocated using the `new` operator:

```c++
int * table;
table = new int [4096];
```

Heap memory is de-allocated using the `delete` operator:

```c++
delete [] table;
```

This is known as dynamic memory allocation/de-allocation
The Heap

heap empty

a = new int [400]
b = new int [200]
c = new int [600]
d = new int [400]

delete [] b

c

James Grimbleby
The heap can become fragmented after many new/delete operations:

Garbage collector can be called to de-fragment heap:
The Heap

Care must be taken to de-allocate heap memory when it is no longer required

```c
void do_something()
{
    int * table = new int [4096];
    // ...
    if (x > 0) return;
    // ...
    delete [] table;
}
```

Failure to de-allocate leads to memory leakage, and ultimately to heap overflow.
Exceptions

Microprocessors normally execute code sequentially

Occasionally the microprocessor is required to suspend execution temporarily while some unusual condition is resolved

Such conditions are termed exceptions

Exceptions may be a reset, various error traps or external interrupts

During exception processing the microprocessor status and the return address are stored on the stack
## PIC24 Status Register

The PIC24 processor status register SR is 16 bits wide:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Carry bit (set if a carry occurred)</td>
</tr>
<tr>
<td>14</td>
<td>Zero bit (set if the result was zero)</td>
</tr>
<tr>
<td>13</td>
<td>Overflow bit (set if an overflow occurred)</td>
</tr>
<tr>
<td>12</td>
<td>Negative bit (set if the result was negative)</td>
</tr>
<tr>
<td>11</td>
<td>Repeat loop active</td>
</tr>
<tr>
<td>10..1</td>
<td>Current processor priority level</td>
</tr>
<tr>
<td>0</td>
<td>Half carry/borrow</td>
</tr>
</tbody>
</table>

C  carry bit (set if a carry occurred)
Z  zero bit (set if the result was zero)
V  overflow bit (set if an overflow occurred)
N  negative bit (set if the result was negative)
R  repeat loop active
I0..I2 current processor priority level
D  half carry/borrow
## PIC24 Exception Vectors

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
<th>Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x0000</td>
<td>Reset: GOTO Instruction</td>
</tr>
<tr>
<td>1</td>
<td>0x0002</td>
<td>Reset: GOTO Address</td>
</tr>
<tr>
<td>2</td>
<td>0x0004</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>0x0006</td>
<td>Oscillator fail trap</td>
</tr>
<tr>
<td>4</td>
<td>0x0008</td>
<td>Address Error trap</td>
</tr>
<tr>
<td>5</td>
<td>0x000A</td>
<td>Stack error trap</td>
</tr>
<tr>
<td>6</td>
<td>0x000C</td>
<td>Math error trap</td>
</tr>
<tr>
<td>7</td>
<td>0x000E</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>0x0010</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>0x0012</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>0x0014</td>
<td>Interrupt vector 0</td>
</tr>
<tr>
<td>11</td>
<td>0x0016</td>
<td>Interrupt vector 1</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>62</td>
<td>0x007C</td>
<td>Interrupt vector 52</td>
</tr>
<tr>
<td>63</td>
<td>0x007E</td>
<td>Interrupt vector 53</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>126</td>
<td>0x00FC</td>
<td>Interrupt vector 116</td>
</tr>
<tr>
<td>127</td>
<td>0x00FE</td>
<td>Interrupt vector 117</td>
</tr>
</tbody>
</table>
Exception Priority

The current processor priority is determined by the bits: I0-I2 in the SR and bit: I3 in the CORCON register

Exceptions only occur when the exception priority is greater than the current processor priority

Interrupts have priority 1 to 7

Soft error traps have priority levels 11 to 12

Hard error traps have priority levels 13 to 15

Hard and soft error traps are non-maskable: they have higher priority than any interrupt
Reset Exception

The reset exception has the highest exception priority

Reset exception is raised at system start-up, by the MCLR line being pulled low (active), or by a `reset` instruction

Any processing in progress at the time of the reset is lost

Processor begins execution at location 0x000000

Locations 0x000000 and 0x000002 must contain a `goto` instruction and the address of the start of the program

The stack pointer W15 is initialised to 0x0800 and the current processor priority is set to 0
Oscillator Fail Trap

Oscillator fail trap is a hard trap with priority 14

The PIC24 system clock can be driven by external clock sources, external resonators or internal RC clocks.

In the event of an oscillator failure, the Fail Safe Clock Monitor will generate an oscillator failure trap.

At the same time it will switch the system clock to the internal fast RC oscillator.

The OSCFAIL flag must be reset before returning from an oscillator fail trap.
Address Error Trap

Address error trap is a hard trap with priority 13.

An address error trap is raised if:
- A word instruction accesses an odd address
- An indirect bit instruction accesses an odd address
- Data is fetched from unimplemented address space
- A literal \texttt{bra} or \texttt{goto} is executed to unimplemented address
- The program counter is modified to unimplemented address

The \texttt{ADDRERR} flag must be reset before returning from an address error trap.
Stack Error Trap

Stack error trap is a soft trap with priority 12

The stack pointer W15 is initialised to 0x0800 on reset

A stack error trap is raised if:

- The stack pointer falls below 0x0800
- The stack pointer exceeds the value in the stack pointer limit register SPL

This trap can be used to prevent software failure resulting from stack overflow

The STKERR flag must be reset before returning from a stack error trap
Math Error Trap

Math error trap is a soft trap with priority 11

PIC24 has an integer divide function

The result of dividing by zero is undefined

A math error trap is raised if an attempt is made to divide by zero

The MATHERR flag must be reset before returning from a math error trap
Exception Processing

Exception processing occurs if its priority exceeds the current processor, and has four identifiable steps:

1. The current instruction is completed

2. The program counter, low byte of SR and bit I3 of CORCON register are stored on stack

3. The current processor priority is set to the exception priority

4. The new program counter value is fetched from the exception vector table
An `RETFIE` instruction at the end of the exception code returns execution to the point where the exception occurred.

`RETFIE` is similar in effect to `RETURN` except that it reloads both the SR and PC from stack.

This returns the processor to the processor priority before the exception occurred.
Exception Processing

During exception processing the return address and the status are stored on the stack:
Re-entrant functions can be interrupted by an exception, and then called from the exception processing routines.

Local variables stored in fixed locations will corrupted when the function is called from the exception routines.

Re-entrant functions must have their variables stored in a stack frame.

Then a new set of local variables will be created when the function is called from the exception routines.
Interrupt Exceptions

Interrupts are used for servicing peripherals requiring infrequent but fast response

An external peripheral raises an interrupt exception by asserting one of the interrupt input lines

This avoids the need to poll peripherals at regular intervals, leading to more efficient use of the processor

Interrupts can only be used provided that the main task is not time-sensitive

Programs making use of interrupts may be difficult to debug
Interrupt Exceptions

PIC24 has five external edge-sensitive interrupt request inputs.

In addition there are a number of internal peripheral interrupts (such as timer expired, adc complete, uart rx, …)

Each external and peripheral interrupt source has been assigned one of the interrupt exception vectors 0-117.

Each external and peripheral interrupt source must be assigned one of the seven priority levels 1-7.

Interrupt sources with the same priority level have a secondary priority dependent on position in the exception vector table.
Multi-Level Interrupt Exceptions

Main task (current processor priority = 0)

Interrupt priority 2

Interrupt priority 1

Interrupt priority 5

CPU = 0

CPU = 2

CPU = 1

CPU = 5
Some processors such as 16F84 have a single active-low interrupt request (IRQ) line:

Microprocessor must determine source of interrupt by polling: interrogate each device in turn
Interrupt Latency

Interrupt latency is the time between an interrupt request and the start of processing the interrupt routines.

Interrupt latency depends on:

1. The clock frequency
2. The maximum number of clock cycles in an instruction
3. The time to save the status, return address and registers
4. The length of any blocks of code in the main task for which interrupts are masked
Serial Interfacing

Serial data protocols are increasingly being used between processor and peripherals because:

Serial data communication requires fewer PCB tracks (typically 1 or 2 rather than 8 or 16)

Serial data communications uses fewer device pins (typically 1 or 2 rather than 8 to 16) allowing smaller packages

Unfortunately serial data communication speed is lower than parallel and is therefore not used for program or data memory

There are two main classes of serial data communications: synchronous and asynchronous
Synchronous Data Comms

In synchronous communication both data transmitter and data receiver are synchronised to the same clock.

The clock may be generated by the transmitter, or may be independent of transmitter and receiver.

![Serial data and clock diagram]

The data is read when the clock signal is high.
Asynchronous Data Comms

In asynchronous communication there is no common clock signal.

Instead the data is sent in groups (frames) at a rate agreed between transmitter and receiver.

Data frames are synchronised by means of a start bit.

Start bit (1)

Stop bit (0)

Idle (0)
Serial Interfacing

Synchronous serial protocols:

I²C  Inter-Integrated Circuit (Philips)
SPI  Serial Peripheral Interface (Motorola)
Ethernet  (Xerox)
USB  (Microsoft)
FireWire  IEEE 1394 (Apple)

Asynchronous serial protocols:

EIA-232:  Electronics Industries Association (aka RS-232)
RS-422:  Balanced version of RS-232
RS-485:  Multi-drop version of RS-422
CAN:  Controller Area Network (Bosch)
Simplex/Duplex

Serial links are characterised by the directions of data flow:

If data flows in one direction only then the link is termed *simplex*

If data can flow in both directions, but only one direction at a time then the link is termed *half duplex*

If data can flow in both directions simultaneously then the link is termed *full duplex*
Multi-Drop

If more than 2 devices are connected by a serial link then this is termed a *multi-drop* system.

Each device must have a unique address or a select input.

In a multi-drop system one device has to be assigned the controller of communication.

This is termed the *master* device; all other devices are termed *slaves*.

The master device is not necessarily fixed, and devices can take turns to act as master.
Inter-Integrated Circuit (I2C) Bus

I²C is a bi-directional, multi-drop, half duplex, 2-wire, synchronous bus.

The two lines are SCL (serial clock) and SDA (serial data).

The data rate is 400 kbit/s.

Devices on the I²C bus can act as masters or slaves.

The master initiates data transfers and generates the clock; the slave is any device addressed by the master.

Arbitration is used to resolve conflicts if more than one master attempts to control the bus.

James Grimbleby  School of Systems Engineering - Electronic Engineering  Slide 108
Inter-Integrated Circuit (I2C) Bus

$+V_{cc}$

SCL

SDA

Device 1

Device 2

Data out

Clock out

Data in

Clock in
Inter-Integrated Circuit (I2C) Bus

One data bit:

SDA

SCL

SCL is always controlled by master

Data valid

Data may change

SDA may be controlled by master or slave (depending in the direction of data flow)
Inter-Integrated Circuit (I2C) Bus

Data is transferred in 8-bit bytes:

A byte may be the address of a slave, or data transferred between master and slave

An address or data byte is sent most-significant bit first
Inter-Integrated Circuit (I2C) Bus

A data transfer is initiated by the master signalling a \textit{start} condition.

This acts as an attention signal to all devices.

Start is defined as a high-to-low transition of SDA whilst SCL is high.
Inter-Integrated Circuit (I2C) Bus

Master then sends an *address* byte of a slave device

This consists of 7-bit slave address and 1-bit data direction

Slave devices compare the address with their own address
Inter-Integrated Circuit (I2C) Bus

If the address matches then the slave sends *acknowledge*

- Slave pulls SDA low immediately after address byte
- Slave releases SDA after clock pulse

Master generates a clock pulse

---

*James Grimbleby*

School of Systems Engineering - Electronic Engineering
Data is now transferred between master and slave; after each byte is sent the receiver sends *acknowledge*

Finally master indicates end of transmission by a *stop* signal

Stop is defined as a low-to-high transition of SDA whilst SCL is high

![Diagram showing SDA and SCL signals with a stop signal](Image)
EIA-232

EIA-232 (also known as RS-232) is a single-ended asynchronous serial protocol

EIA-232 is suitable for low bandwidth communications (20 kb/s max) over short distances (25 m max)

Typical baud rates are 300, 2400, 9600, 19200 and 38400; the data rate is somewhat less than the baud rate

![Voltage levels diagram]

- Logic 0: +15V
- Logic 1: -15V

James Grimbleby  
School of Systems Engineering - Electronic Engineering  
Slide 116
EIA-232

Start bit (low)

Idle (high)

Data bits (Usually 8)

Detect 16 cycles of 8 cycles of Sample
start bit edge

Stop bit (high)

8 16 16 16 16 16 16 16 8

Detect start bit edge

8 cycles of 16\times clock

Sample data

16 cycles of 16\times clock
Length of EIA-32 data frame is 10 bits (8 data bits + start and stop bits)

Clock accuracy must be such that sample times are accurate to better than 1/2 bit over 10 bits → 5%

Receiver and transmitter clocks use crystal oscillators and are typically accurate to better than 100 ppm → 0.01%

Typical crystal oscillator frequency is 2.4576 MHz: division by $2^n$ generates a $16\times$ clock for all standard baud rates

$16\times$ clock for 9600 baud is 153.6 kHz which is $2.4576/16$ MHz
The EIA-32 standard defines 25 different signals including data transmit and receive, control and ground.

Of these at most 5 are used in embedded systems:

- tx  data transmit
- rx  data receive
- rts ready to send
- cts clear to send
- gnd ground reference

tx and rx carry data; rts and cts are used to control the data flow.
EIA-232

Micro system

Peripheral device

DB9S Connector:

1  6
2  7
3  8
4  rts
5  cts
6  gnd
7  rx
8  tx
9  gnd

James Grimbleby
School of Systems Engineering - Electronic Engineering
Slide 120
RS-422

RS-422 has a similar data frame to EIA-232 but uses low-voltage differential drivers and differential receivers.

This makes possible much higher data rates (10 Mbit/s) over longer distances (1200 m).

It is also much less susceptible to electrical interference than EIA-232.

RS-485 is a multi-drop version of RS422 and is commonly used in industrial process-control applications.
Integer Numbers

Integer values are almost universally stored and processed in natural binary (or signed binary) form.

Normally 8-bit and 16-bit integers are available; more powerful processors also support 32-bit integers:

- **8-bit natural binary**: 0 → 255
- **8-bit signed binary**: -128 → 127
- **16-bit natural binary**: 0 → 65535
- **16-bit signed binary**: -32768 → 32767
- **32-bit natural binary**: 0 → 4294967295
- **32-bit signed binary**: -2147483648 → 2147483647

Arithmetic using integers is exact.

James Grimbleby
**Integer Numbers**

Integers stored in a finite word are cyclic, that is arithmetic operates modulo $m = 2^n$ (where $n$ is the word size).

For 16-bit integers arithmetic operates modulo 65536:

$$40000 + 12000 \rightarrow 52000$$
$$40000 + 36000 \rightarrow 10464 \quad (76000 \mod 65536)$$

In the second example overflow has occurred.

When overflow occurs in addition the result is smaller than either of the operands.
Fractional Binary Numbers

8-, 16- and 32-bit words can be regarded as fractional binary numbers (= integer value / $2^n$)

Fractional binary numbers represent 0.0 to slightly less than 1.0

16-bit fractional binary numbers:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Fractional Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000000000000</td>
<td>0.0</td>
<td>0 / 65536</td>
</tr>
<tr>
<td>0000000000000001</td>
<td>0.000015259</td>
<td>1 / 65536</td>
</tr>
<tr>
<td>0000000000000010</td>
<td>0.000030518</td>
<td>2 / 65536</td>
</tr>
<tr>
<td>..................</td>
<td>..........</td>
<td>..................</td>
</tr>
<tr>
<td>1111111111111111</td>
<td>0.999984741</td>
<td>65535 / 65536</td>
</tr>
</tbody>
</table>

Fractional binary numbers can be signed or unsigned
Fixed-Point Binary Numbers

8-, 16- and 32-bit words can be regarded as fixed-point binary numbers (= integer value / $2^i$ where: $0 < i < n$)

The position of the point is decided by the programmer

For example, 16-bit fixed point binary numbers, with the point between bits 7 and 8 can represent:

- $00000000.00000000$ = $0.0$ (0 / 256)
- $00000000.11111111$ = $0.9960938$ (255 / 256)
- $00000001.00000000$ = $1.0$ (256 / 256)
- $11111111.00000000$ = $255.0$ (65280 / 256)
- $11111111.11111111$ = $255.9960938$ (65535 / 256)
Fixed-Point Binary Arithmetic

Fixed point values are their equivalent integer values divided by $m = 2^i$:

\[
\begin{align*}
\text{Integer } j & \quad \text{Fixed-point } x = j/m \\
\text{Integer } k & \quad \text{Fixed-point } y = k/m
\end{align*}
\]

Addition: \[z = (j + k)/m = j/m + k/m = x + y\]
Subtraction: \[z = (j - k)/m = j/m - k/m = x - y\]

Multiplication: \[z = (j \times k)/m = m \times (j/m \times k/m) = m \times x \times y\]

From this it is clear that the normal add and subtract operations can be used for fixed-point numbers, but the result of multiplication must be divided by $m$. 
Characters

Characters are normally 8-bits and use the ASCII code:

<table>
<thead>
<tr>
<th>ms 3 bits</th>
<th>ls 4 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>A B C D E F</td>
</tr>
<tr>
<td>nul</td>
<td>If</td>
</tr>
<tr>
<td>1</td>
<td>esc</td>
</tr>
<tr>
<td>sp</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>3</td>
<td>@</td>
</tr>
<tr>
<td>4</td>
<td>P</td>
</tr>
<tr>
<td>5</td>
<td>`</td>
</tr>
<tr>
<td>6</td>
<td>p</td>
</tr>
</tbody>
</table>
Booleans

C++ has a built-in Boolean type: bool, which can have values true or false

C does not have a boolean type, but one can easily be created by using macros:

```c
#define bool int
#define false 0
#define true (! false)
```

or by using enumerations:

```c
enum bool {false = 0, true = 1};
```
Floating-Point Numbers

It is unusual for embedded microprocessor systems to use floating-point numbers.

This is because the quantities being processed normally cover a limited range of values.

Floating-point arithmetic is much more computationally expensive than integer arithmetic.

Most compilers and floating-point libraries support the IEEE 32-bit single-precision format (23-bit mantissa, 8-bit exponent).

Floating-point arithmetic involves truncation or rounding and is therefore not exact.
Floating-Point Numbers

Binary floating-point codes can represent very large and very small values.

Floating-point numbers consist of two fields, the mantissa $M$ and the exponent $E$

$$X = M \times 2^E$$

The relative precision of a floating-point number is determined by the number of bits in the mantissa $M$

The range is determined by the number of bits in the exponent $E$
IEEE Floating-Point Formats

The IEEE standard defines the format of single (32 bit), double (64 bit) and extended (80 bit) precision floating-point numbers.

Single precision (32 bit):

31 30 ........ 23 22 ................................................................. 0

1 bit sign  8 bit exponent  23 bit mantissa

Mantissa is 23-bit unsigned fixed-point binary

Exponent is 8 bit excess 127 binary
IEEE Floating-Point Formats

Mantissa 23-bit + hidden bit:

\[
\begin{align*}
1.00000 \ldots 00000 & = 1 \text{ (decimal)} \\
1.11111 \ldots 11111 & \approx 2 \text{ (decimal)}
\end{align*}
\]

\uparrow \quad 23 \text{ bits of IEEE format}

\uparrow \quad \text{Hidden bit (always 1)}

A change in the least-significant bit is equal to:

\[
2^{-23} = 1.2 \times 10^{-7}
\]

so that IEEE 32-bit numbers have a relative precision of about 7 decimal digits
IEEE Floating-Point Formats

Exponent 8-bit excess 127:

\[
\begin{align*}
1 - 127 &= -126 \\
254 - 127 &= +127
\end{align*}
\]

Floating-point range:

\[
\begin{align*}
1 \times 2^{-126} &= 1.2 \times 10^{-38} \\
2 \times 2^{+127} &= 3.4 \times 10^{+38}
\end{align*}
\]

The special case of 8-bit exponent all 0s is used to represent the floating-point value 0.0

The special case of 8-bit exponent all 1s is used to represent not-a-number
PIC18 is optimised for processing single bits or 8-bit words, and this is reflected in the CCS compiler word sizes:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8-bit</td>
</tr>
<tr>
<td>short int</td>
<td>1-bit</td>
</tr>
<tr>
<td>int</td>
<td>8-bit</td>
</tr>
<tr>
<td>long int</td>
<td>16-bit</td>
</tr>
<tr>
<td>int32</td>
<td>32-bit</td>
</tr>
<tr>
<td>float</td>
<td>32-bit</td>
</tr>
</tbody>
</table>

Operations involving long ints and floats are performed by subroutines.
### C30 C Compiler Targeting PIC24

PIC24 is optimised for processing single bits, 8-bit bytes or 16-bit words:

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char:</td>
<td>8-bit</td>
</tr>
<tr>
<td>short int:</td>
<td>16-bit</td>
</tr>
<tr>
<td>int:</td>
<td>16-bit</td>
</tr>
<tr>
<td>long int:</td>
<td>32-bit</td>
</tr>
<tr>
<td>long long int</td>
<td>64-bit</td>
</tr>
<tr>
<td>float:</td>
<td>32-bit</td>
</tr>
<tr>
<td>double</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

Operations involving long ints and floats are performed by subroutines.
Multi-Precision Operations

It is often necessary to process data words that are larger than can be operated on by a single instruction.

PIC18 instructions only operate on 8-bit words.

PIC24 instructions operate on 8- and 16-bit words.

Multi-precision arithmetic uses a sequence of basic instructions on existing data types.

Operations on multi-precision numbers are normally incorporated into the program code as subroutines.
Multi-Precision Addition

\[ Ams += Als + Bms = Bls \]

Overflow?

Yes

Ams ++

No

Ams += Bms
Multi-Precision Addition

\[ \text{Ams} + \text{Als} = \text{Bms} + \text{Bls} \]

**PIC16:**
\begin{align*}
\text{movf} & \quad \text{Bl} s, 0 \\
\text{addwf} & \quad \text{Al} s, 1 \\
\text{btfsc} & \quad \text{status}, 0 \\
\text{incf} & \quad \text{Ams}, 1 \\
\text{movf} & \quad \text{Bms}, 0 \\
\text{addwf} & \quad \text{Ams}, 1 \\
\end{align*}

**PIC24:**
\begin{align*}
\text{mov} \ .w & \quad \text{Al} s, W0 \\
\text{mov} \ .w & \quad \text{Ams}, W1 \\
\text{mov} \ .w & \quad \text{Bl} s, W2 \\
\text{mov} \ .w & \quad \text{Bms}, W3 \\
\text{add} \ .w & \quad W2, W0, W0 \\
\text{addc} \ .w & \quad W3, W1, W1 \\
\end{align*}
Multi-Precision Addition

Multi-precision increment += operator:

```c
struct xlong{
    long int ms, ls;
};

void operator += (xlong & a, const xlong & b) {
    long int c = a.ls;
    a.ls += b.ls
    if (a.ls < c) // overflow
        a.ms++;
    a.ms += b.ms;
}
```
Multi-Precision Addition

The multi-precision + operator can now be defined in terms of the += operator:

```c
void operator += (xlong & a, const xlong & b);

xlong operator + (const xlong & a, const xlong & b)
{
    xlong c = a;
    c += b;
    return c;
}
```
Multi-Precision Addition

Using a C++ class:

class xlong {
    private:
        long int ms, ls;
    public:
        xlong {ms = 0; ls = 0;}
        void operator += (const xlong & b);
        . . . .
    }

void xlong::operator += (const xlong & b) {
    long int c = ls;
    ls += b.ls
    if (ls < c) // overflow
        ms++;
    ms += b.ms;
}
Multi-Precision Comparison

Ams > Als > Bms

Ams == Bms

Ams > Bms = true
false

Als > Bls = true
false
bool operator > (const xlong a, const xlong b)
{
    if (a.ms == b.ms)
        return (a.ls > b.ls);
    else
        return (a.ms > b.ms);
}
Multi-Precision Bit Shifting

Shift left:

```c
void operator <<= (long & a, int s)
{
    a.ms = (a.ms << s) | (a.ls >> (32 - s));
    a.ls <<= s;
}
```
Multi-Precision Bit Shifting

Shift right:

\[ \text{void operator} \ggg (xlong & a, \text{ int s}) \]
\[
\{
    a.\text{ls} = (a.\text{ls} \gg s) | (a.\text{ms} \ll (32 - s));
    a.\text{ms} \ggg s;
\}

James Grimbleby  School of Systems Engineering - Electronic Engineering  Slide 145
Multi-Precision Bit Shifting

The normal >> and << operators can now be defined in terms of the >>= and <<= operators. For example:

```c
void operator >>= (xlong & a, int s);
void operator <<= (xlong & a, int s);

xlong operator >> (xlong & a, int s)
{
    xlong c = a;
    c >>= s;
    return c;
}
```
Product Development Cycle

10% Requirements Analysis

10% Specification

10% Design

10% Documentation

20% Coding

20% Testing and Debugging

20% Maintenance

Approximate effort for each stage
Requirements Analysis

This phase is normally conducted by marketing personnel.

It may be in response to a request from a customer, or a perceived marketing opportunity.

It involves matters such as:

- Market price
- Development costs
- Development time scale
- Production costs
- Product lifetime
Specification

This phase defines the external functions of the system.

For example, the specification for a microprocessor-controlled positioning system might contain:

- List of commands (move, stop, index etc)
- Packaging
- Voltage and current levels for the inputs and outputs
- Temperature limits
- Electromagnetic compatibility
- Power supply requirements
Design

There will usually be several possible design solutions which can meet the specification.

The first step is to select the best design solution.

Then the problem is broken down into a number of hardware and software modules.

The exact function and interface of the modules should be specified.

This allows teams of hardware engineers and programmers to work on separate hardware and software modules.
Documentation

Documentation should be generated before coding, not as is often the case at the end of the project.

This ensures that the coding is consistent with the required system behaviour.

Documentation generated after coding will reflect any idiosyncrasies in the coding.
Coding

Coding is where the software modules are converted into a programming language.

The language may be either:

- a microprocessor-specific assembly language
- a microprocessor-independent high-level language

The use of a high-level language makes it easier for teams of programmers to work on separate modules.
Testing and Debugging

It is extremely rare for an embedded system to operate correctly first time.

Debugging is the process of determining the reasons for failure and correcting them.

Once the system appears to be at least partly working it must be rigorously tested.

Testing involves challenging the system with the most difficult conditions that it will encounter in practice.

If it fails under testing then the debugging process must be resumed to find out why.
Maintenance

In any large program there are bound to be errors which are not found during the testing phase.

These may become apparent months or years after the original coding.

It is also likely that there will be requests for changes in specification, requiring code modifications.

The cost of maintenance is highly dependent on the level of documentation.

A poorly documented system may be almost impossible to maintain.
Software Development

There are four distinguishable approaches to software development:

Linear: Start at beginning and continue to end

Evolutionary: Start from some small but complete initial program, and add features as required

Bottom up: Modules to fulfil certain tasks are written; these are then combined to produce complete program

Top down: Write program to perform overall task, with sub-tasks replaced by stubs; then write code for stubs
Assembler Code

In assembly code the machine instruction codes are replaced by meaningful mnemonics.

PIC24 example: EB0100 is replaced by: clr.w W2

Mnemonics are converted to machine codes by an assembler program which also reports any errors.

Large programs written in assembly language are expensive, error-prone and difficult to maintain.

Assembly language programming is used for small programs and where for a good high-level language compiler is not available.
Assembler Code

Code to toggle the Port A lines:

```
// Title   : PIC Lab Demo Program
// LastEdit: 22 July 2007
// Author  : J. B. Grimbleby

.equ tris, 0x02c0
.equ port, 0x02c2

clr.w tris:
mov.w #0xff, W0
mov.w W0, port
loop: mov.w port, W0
com.w W0, W0
mov.w W0, port
bra loop
```
Code to toggle the Port A lines:

```assembly
// Title : PIC Lab Demo Program
// LastEdit: 22 July 2007
// Author : J. B. Grimbleby

.equ tris, 0x02c0
.equ port, 0x02c2

clr.w tris:
  mov.w #0xff, w0
  mov.w w0, port
loop:
  mov.w port, w0
  com.w w0, w0
  mov.w w0, port
  bra loop
```

James Grimbleby
High-Level Languages

High-level languages are designed for ease of use and do not correspond directly to machine codes.

Examples of high-level languages are BASIC, ADA, C++ and JAVA.

Programs written in high-level languages are normally transportable.

Writing software in a high-level language is faster and less error-prone than writing in assembler code.

High-level languages can either be interpreted or compiled.
An interpreter is a program resident on the microprocessor system which operates directly on the high-level program.

Each high-level instruction is translated and executed as the interpreter comes to it.

If the program contains a loop with 1000 iterations, then the instructions in the loop will have to be translated 1000 times.

An interpreted program may typically run up to 10 times slower than a compiled program.

Because the interpreter retains control good diagnostics are usually available and the program cannot run amok.
Compilers translate high-level language programs into either assembly code or directly into machine code.

Compilers perform a degree of error-checking and can therefore detect some programming mistakes.

Compilation need only be performed once.

A good optimising compiler can produce code as efficient as well-written assembler code.

Compilers for embedded microprocessor systems do not reside on the target system and are termed cross-compilers.
The BASIC Language

BASIC was the first high-level language to be interpreted.

It is an old-fashioned language based on FORTRAN.

Loop and conditional constructs are primitive; subroutines cannot have parameters.

Data types are limited, and cannot be created by the programmer.

Basic is almost always interpreted rather than compiled.

Some microprocessors have a built-in BASIC interpreter.
The ADA Language

ADA is a modern modular language based on Pascal, which in turn was based on ALGOL.

It was designed specifically for programming embedded microprocessor systems.

ADA code is intended to be readable, maintainable and secure.

It is a real-time language providing facilities for multi-tasking and interfacing with hardware.

ADA is a large and complex language and has yet to be widely adopted for programming microprocessor systems.
The Java Language

The Java programming language is a general-purpose object-oriented concurrent language.

Its syntax is similar to C/C++, but it omits many of the features that make C++ inefficient or unsafe.

Java programs are compiled to an abstract intermediate code (byte code) that runs on a Java virtual machine.

This is in effect an interpreter program that runs on the target processor and executes the intermediate code.

Java execution is not particularly fast, and at present it is used mainly in Web applications.
The C Language

The most popular high-level language for programming embedded microprocessors is C or C++

C has high level constructs and at the same time supports the low level operations essential for embedded systems.

It is rarely necessary to resort to inserting blocks of assembly code in C.

Unlike most other high-level languages, C programs do not normally require a large operating system.

C cross-compilers are available for almost all microprocessor types.
The C++ Language

C++ is an extension of C which supports data abstraction and object-oriented programming.

C++ contains C as a subset.

Some features of C++ may be demanding in terms of memory and may slow program execution.

Embedded C++ is a subset of C++ targeted at embedded systems.

Embedded C++ is a compromise between the efficiency of C and the power of C++.
Hardware is accessed in C/C++ using pointers

If the hardware is byte organised then char pointers are used.

If the hardware is word organised then int pointers are used.

Example: an 8-bit input port memory-mapped to location 0x02c2:

```c
#define port ((char *) 0x02c2)
```

Thus `port` is a char pointer whose value is the address of the bus device.
Hardware Access in C/C++

The port is accessed by the use of the indirection operator *:

```c
char p;
p = *port;
```

In this example, $p$ is assigned the value of memory location 0x02c2 which happens to be the input port.

C++ contains a keyword volatile:

```c
#define port ((volatile char *) 0x02c2)
```

This prevents the compiler using a register to mirror a port.
Hardware Access in C/C++

Code to toggle the Port A lines:

```c
// Title : PIC Lab Demo Program
// LastEdit: 22 July 2007
// Author : J. B. Grimbleby

#define tris ((volatile char *) 0x02c0 )
#define port ((volatile char *) 0x02c2 )

void main() {
    *tris = 0;
    *port = 0xff;
    for (;;) {
        *port = ~*port;
    }
}
```
#define tris ((volatile char *) 0x02c0 )
#define port ((volatile char *) 0x02c2 )

void main() {
    *tris = 0;
    *port = 0xff;

    for (;;)
        *port = ~*port;
}

Safety-Critical Systems

A safety-critical system is one where malfunction could lead to injury or death

Examples of safety-critical systems include:

- Automotive anti-lock braking systems
- Medical infusion pumps
- Traffic lights
- Nuclear reactor criticality control
- Aircraft flight control
- Chemical plant control

It is essential to follow a strict design methodology for safety-critical systems
Safety-Critical Systems

As far as is possible safety-critical systems should be fail-safe

Safety-critical systems should also be fault-tolerant and complete failure should occur only as a result of multiple component failures

The probability of complete system failure must be estimated, and considered alongside the possible consequences

Failure analysis: identify failure modes, and their probability based on known component failure mechanisms

Hazard analysis: list possible hazards that may occur as a result of system failure, and estimate their severity
Safety-Critical Systems

Failure probability

Frequently
Probably
Occasionally
Infrequently
Exceptionally
Impossible

Hazard Severity

Acceptable
Unacceptable

Negligible Minor Serious Critical Catastrophic

James Grimbleby  School of Systems Engineering - Electronic Engineering  Slide 173
Fail-Safe

As the name implies a fail-safe system should fail in a safe configuration:

If an electric power-assisted steering system fails it should still be possible to steer the vehicle (without power assistance)

If a traffic light controller fails then all the lights should go to red (thus stopping all traffic and pedestrians)

Fail-safe is often achieved by using additional hardware to monitor the primary microprocessor, and to take appropriate action if necessary

For example a watchdog can be used to provide fail-safe operation
Hardware Failure

Most hardware faults are normally immediately apparent

Electronic components are now extremely reliable and unless they are used outside the manufacturer's recommended conditions they are very unlikely to fail in operation

Some components "wear out" but normally have predictable lifetimes (for example electrolytic capacitors)

Environmental factors such as excessive temperature, vibration can cause temporary or permanent component malfunction
A soft error is a transient hardware error that occurs in a component that is otherwise functioning satisfactorily.

For example dynamic memory uses small capacitors to store data and a single bit of data can occasionally be corrupted by the charge generated by alpha particles.

Soft errors may lead to corruption of data being processed and this can have serious consequences.

Corruption of a subroutine return address stored on the stack would probably lead to complete program failure.
“Hard” Hardware Failure

A hard error is a permanent malfunction of a component, usually as a result of thermal or electrical damage.

In a dynamic memory one of the capacitors used to store data may become short-circuited resulting in the data stored always being 0.

Hard errors are more serious than soft errors and unless precautions are taken will lead to complete system failure.

The effects of both hard and soft errors can be reduced by designing fault tolerance into the system.
Fault-tolerant hardware

Fault tolerance allows a system to continue to operate even when some components have failed.

Single point failure means that the failure of a single component can cause complete system failure.

Redundancy is used to avoid single point failure and therefore provide fault-tolerance.

Failure in redundant systems only occurs if several components fail.

There are two types of redundancy: hardware redundancy and data redundancy.
Hardware Redundancy

In a redundant system each function is performed by several components.

Failure of a single component, or even of several components, will not cause system failure.

For example:

Connecting two decoupling capacitors in parallel provides redundancy if one of them fails open-circuit.

Connecting two diodes in series provides redundancy if one of them fails short-circuit.

The use of three separate flight control computers in an aircraft provides redundancy if one computer fails.
Hardware Redundancy

This arrangement will continue to operate if 2 of the 5 redundant systems fail.
Data Redundancy

Errors in read/write memory can be detected, and possible corrected, by the use of data redundancy.

A parity bit added every time a byte of data is stored can be checked when data is retrieved.

A single bit error in the data will change the parity and can therefore be detected.

The use of a simple parity bit does not allow the error to be corrected and the data will be lost.

Parity error can be used to raise an exception, reset the processor or put the system in a fail-safe configuration.
Data Redundancy

By adding further bits to the code it can be made error-correcting.

If a single bit error occurs then the original data can be recovered by suitable hardware.

The Hamming error-correcting code adds $c$ error-correction bits to the $k$ data bits, where:

$$2^c \geq c + k + 1$$

Most embedded processor memories are byte-organised, so that $k = 8$ and $c = 4$. 
Watchdog Timer

A very simple and effective method for detecting hardware and software faults is the watchdog timer.

This is a timer that must be reset on a repetitive basis by the properly-executing program.

If the timer is not reset within a predetermined time then it generates a reset or NMI exception to the microprocessor.

It may also force any outputs from the microprocessor to a safe configuration.

Many modern microprocessors have on-chip programmable watchdog timers.
Watchdog Timer

Address decoder responds to a single address and triggers the re-triggerable monostable, setting /RESET high (inactive)
Software Failure

Experience has shown that software failure is most serious threat to the correct operation of embedded microprocessor systems.

It has proved to be impossible in practice to write computer code of any complexity that is error free.

Unlike hardware errors the effect of software errors is difficult to predict: the effect may be benign leading to no observable consequences, or catastrophic.

It is estimated that 200 people have been killed or injured in industrial control accidents as a result of failure of microprocessor software.
Examples of Software Failure

The Gemini V re-entry vehicle splashed down 100 miles off course because the programmers forgot that the Earth rotated during re-entry.

A London Docklands automatic train was left hanging over a roadway with its doors open because the train had gone into "fail-safe" mode as a result of a software bug.

In 1982 a software bug caused a huge (3 kton) explosion on the trans-Siberian gas pipeline but surprisingly there were no casualties.

The first launch of the Ariane 5 rocket failed because of an exception raised during conversion of a 64-bit floating point number into a 16-bit integer.
Examples of Software Failure

The Therac-25 medical linear accelerator was responsible for six accidents involving massive overdoses of radiation, three of which lead to deaths.

Updated software in a vehicle production line caused an industrial robot to throw a car on top of the safety inspector's hut (which was fortunately not occupied at the time).

A software error at the accident-prone Sellafield nuclear reprocessing plant was blamed for causing radiation safety protection doors to be opened erroneously.

America nearly launched a nuclear strike on the USSR when software incorrectly identified the rising moon as a Russian rocket.

James Grimbleby
Software Engineering

In the 1980s writers of computer software coined the term Software Engineering

In fact there is very little evidence of any real engineering discipline being applied to writing computer software

Failure rates that would be considered laughable in any other field of engineering are tolerated in computer software

Little effort has been devoted to quantifying software failures as a preliminary to developing more reliable software writing methods

From the earliest days progress in software writing has been dictated more by fashion that by science
Software Engineering

The limited data available suggests that over the last 20 years the number of errors per 1000 lines of code has remained almost unchanged at around 5

It is often said that the choice of computer language is crucial in containing software errors

This is not supported by the evidence: Ada was developed specifically to improve reliability but Ada code has a similar error rate to C

It surely is a poor reflection on the development of software writing techniques that there are at least 100 different computer languages in current use
A common belief is that breaking code up into small components improves reliability.

This has been shown **not** to be the case for a number of different computer languages.

The optimum size for components seems to be around 200 lines, which is much bigger than most programmers would use in practice.

It is also considered that reusing code improves reliability.

Again, this has been shown in practice **not** to be the case.
Other fashions in software development have been structured programming, CASE tools, formal methods and object-oriented design.

None of these have lead to a significant improvement in software reliability.

Some studies indicate that formal methods, used in conjunction with a strict testing regime, may bring the error rate down to less than 1 error per 1000 lines of code.

Contrast this with the fact that some large programs contain more than a million lines of code!
C is widely used for embedded processors in the motor industry, often in safety-critical applications.

However C has a number of features that may lead to poor code reliability.

The Motor Industry Software Reliability Association (MISRA) has produced a set of rules for writing safety-critical software.

These are intended to lead to portability, good program structure and correct coding.

The MISRA rules essentially define a subset of C.
MISRA C

MISRA C consists of 127 rules, for example:

Mandatory rules
1. All code to be ISO 9899 standard C
50. Floating-point variables never tested for equality
56. The goto statement must never be used
57. The continue statement must never be used

Advisory rules
10. Sections of code should not be commented out
47. Operation should not rely on C's operator precedence
82. Functions should have a single point of exit
Embedded Microprocessor Systems

© James Grimbleby 20 October 2008