Software Engineering Basics
Why am I doing this tutorial?

• To make life easier for the tutors
  – And also you
  – Mostly the tutors

• To give you guys a slightly better understanding of programming
  – In my experience it is not taught as well as it could be in mechatronics
Software Engineering

• What is the role of a software engineer?
A brief history of software engineering

• I am not a historian, google it.
• Official history starts with World War 2 code-breaking and artillery calculation machines
• A bunch of other stuff
• But we aren’t software engineers, we’re mechatronic engineers and we like cool machines......
Maillardet’s Draughtsman-Writer

Built in 1800 by Henri Maillardet, a Swiss mechanician of the 18th century who worked in London producing clocks and other mechanisms

Able to write/draw about a dozen different poems/pictures
Stores programs on the brass cams (disks) underneath it. The irregular edges of the disks control the movement of the hand.
Enfant chéri des dames,
J'ésuis en tout pays
Fort bien avec les femmes,
Même avec les maris.

Signé par l'Automate de Maillarder.
• A tonne of planning and design would have gone into that clockwork robot before anything was manufactured

• From ~1960s to ~1980s it was recognised that software engineering was really hard and not as productive as it could be – the “software crisis”
  – Cost and time overruns, projects never being finished, rapidly changing hardware, increasing complexity of problems, craptacular programming tools (punch cards, assembly)

• You can get by with a lot less planning today
  – but it is not advisable!
Software Design Principles

- Divide and conquer
- Close the loop as soon as possible
- Always save a backup
- Always save another backup
- Develop the system incrementally and grow it organically
- Test continuously
- Debug continuously
- Don’t get ambitious
Useful Laws

• Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

• Parkinson’s Law: Work expands so as to fill the time available for its completion.

• Murphy’s Law: If there are two ways of doing something, and one of them will cause catastrophe, someone will do it that way.

• Sod’s Law: Anything that can go wrong, will go wrong.
The Engineering Problem-Solving Approach

• Define the problem
• Plan the solution
• Implement the solution
• Test the solution
• Modify the solution
1. Define the problem

• Probably the hardest problem of all, else we’d all be making millions of dollars inventing things

• Mostly (but not entirely) done for you in the assignment specification.
  – READ THE SPECIFICATION
  – READ THE SPECIFICATION AGAIN
2. Plan the solution

• Block diagrams – very useful if done right
• Flow charts – pretty terrible to do formally
  – Pretty fantastic to do informally
• Pseudocode – fantastic.
Block Diagrams

• High-level overview
• Less detail than other methods
• Clear distinction needs to be made between hardware and software if you’re including both
Flowcharts

• One direction of flow.
  – Left to right or top to bottom
  – Top to bottom is usually better

• One line of flow per chart
  – But sometimes multiple charts per program eg: interrupts

• Flow in straight lines

• Convey the flow of logic

• Use subprograms if it gets too big and complicated
## Flowchart symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Start/end" /></td>
<td>Start/end</td>
<td>An oval represents a start or end point.</td>
</tr>
<tr>
<td><img src="image" alt="Arrows" /></td>
<td>Arrows</td>
<td>A line is a connector that shows relationships between the representative shapes.</td>
</tr>
<tr>
<td><img src="image" alt="Input/Output" /></td>
<td>Input/Output</td>
<td>A parallelogram represents input or output.</td>
</tr>
<tr>
<td><img src="image" alt="Process" /></td>
<td>Process</td>
<td>A rectangle represents a process.</td>
</tr>
<tr>
<td><img src="image" alt="Decision" /></td>
<td>Decision</td>
<td>A diamond indicates a decision.</td>
</tr>
</tbody>
</table>

**Subprogram:** Represents another, separate flowchart
An example using subprograms
Using Symbols Properly

• Be consistent in your yes/no arrows
  – Eg: always down for yes, always right for no

• Be consistent in the direction of flow eg: always right and down for forward, always up and left for back

• When looping back, loop into the arrow running down into the box, not into the box itself

• Do everything in straight lines with 90-degree turns
  – No curves

• Use enough text in the boxes, but no more.

• One line into each box. At most, two lines out.
Everyone’s favourite game:

What is wrong with these flowcharts?
• TOO SMALL
• Nearly all of you are guilty of this
• It sickens me.
• Don’t write your flowchart on a postcard, use the whole page if it’s available
Start

READ N

M = 1
F = 1

F = F * M

IS M = N?

NO

M = M + 1

YES

PRINT F

END
• Bad variable names
• This is not allowed in code, and it’s not allowed in flowcharts
• Too much colour difference between cells
• Too much sideways movement
• First loopback arrow goes into the cell, not the arrow
• Otherwise good consistency in selection arrows
• Wrong symbols
• Too many parallel lines of flow
• Arrows go into boxes, not into the line of flow going into them
• Missing an arrow in the bottom-left
Pseudocode

- BEGIN....END
- IF (condition)...ENDIF
- WHILE (condition)...ENDIF
- **Subprogram**
- Use subprograms if it gets too big and complicated
2. Plan the solution

- There is more to the solution than just the technical aspect
  - Timeline
  - Allocation of tasks
  - Set deadlines and milestones and meet them

- How does a project become a year late?
  - One day at a time.
3. Implement the Solution

• Implement a skeleton of the program first – **block diagrams are your friend**
• Get all the interrupts and functions set up as stubs
  – A stub is an empty function that does nothing but stops code breaking
  – Maybe it will send useful debugging info
• Fill them in one by one
• Interrupt-driven serial is probably the most useful thing to set up first, as it lets you:
  – Control what functions you run (based on input from the PC)
  – Know what is happening inside the microcontroller, and develop filtering algorithms (based on output displayed in Hyperterminal)
3. Implement the Solution (part 2)

- Remember earlier principles
  - Divide and conquer
  - Close the loop as soon as possible
  - Debug, test and backup
  - Grow the program organically
**Things Not To Do**

- **Do not** start coding without some sort of flowchart or pseudocode or planning of what you are trying to do.

- **Do not** write a bunch of code, paste it into the solution, then ask the tutors for help when it doesn’t work
  - We hate this

- **Do not** build entire modules in different programs, getting them to work independently, then throwing them all together.
  - You can very easily end up with a Frankenstein’s monster-like creation of misfitting parts that do not play nicely together. It will destroy all the things you love and drive you to madness.

- **Do not** leave the hardware to the last week. Do it first
More Things Not To Do

• **Do not** be lazy and mooch off the other guys in your group.

• **Do not** be hyperenthusiastic and do it all yourself

• **Do not** leave it to the last minute

  – I am willing to bet $$$cash money$$$ that people will leave it to the last minute and there will be several groups in the lab the night before it is due
4. Test the solution

• Normally done in the half hour before marking
• REALLY SUPER IMPORTANT
• According to Brooks, should take up HALF your time
  – Distinguishes between
    • Component and early system testing
    • Full system testing
• When we mark you, we will have a checklist of things to tick off.
  – Make your own checklists based on the assignment specification
Software Development Time Breakdown

- Planning: 33%
- Component Testing: 25%
- System Testing: 25%
- Coding: 17%

Source: F. Brooks, “The Mythical Man-Month” 1975
5. Modify the solution

• Fix what’s broke
• Go back and revise different steps
• Later on you’ll probably need to revise your ambitions
If you only remember one thing

• Proper planning and preparation prevents poor performance
  – Don’t just code. Write stuff on paper and then code
  – Even when you are in a hurry
  – Especially when you are in a hurry