Problem Definition, Program Design
Problem Definition

How does one

- Formulate tasks as programs?
- Combine short programs into a working system?
- Ensure that the system does what it should do?
Stages of Software Development

- Problem Definition
  - formulation of task in terms of the requirements it places on the software, µP and other hardware

- Software ⇔ Hardware Design
  - Outline of computer program to perform the defined task
  - Hardware to execute it on

- Coding

- Verification (Testing)

- Debugging
  - Simulator, oscilloscope, logic analyser, ICE

- Maintenance
  - Respond to changed requirements
  - Never know when you have found the last bug.

- Documentation (everywhere!)
  - Comments, STD, DFD, memory map, etc.
Stages of Software Development

– but –

Note that software design and development is *not a linear activity!*
Q: Which Statement is a Self-Fulfilling Prophecy?

1. We’d better start coding straight away because we’re going to have a lot of debugging to do.

2. We won’t need much time for testing because we’re not going to find many defects.

3. We’ve put so much thought into the requirement and architecture design that I can’t think of any major problems we’ll run into during coding or debugging.

A: All of them, but the last is preferred...
Problem Definition

- You can’t solve a problem unless you know what it is!
Block Diagram or Context Diagram

- Useful for preliminary definition of a system.
- Example: μP-controlled battery charger.
Problem Definition Checklists

Precise, comprehensive definition of the task is essential. This is often the most important phase of a development project, and deserves to have time invested in it.

During the definition phase, the designer must consider
1. Inputs
2. Outputs
3. Processing
4. Error handling
5. Human factors (human-machine communication)

Often, this information must be extracted from the client!

Leads to the Requirements Document.
1. Inputs

- List all the inputs that the \( \mu \)P may receive.
- For each input, consider
  - What is the exact form of the signal that the \( \mu \)P will receive?
  - When is it available; how does the \( \mu \)P know that it is available?
  - For how long is it available?
  - How often does it change; how does \( \mu \)P know that it has changed?
  - Block of data, or a sequence? Is the order important?
  - What should be done if the data contains errors? Incorrect data? Out of sequence? Too much? None?
  - Is the input related to any other inputs or outputs?
  - Format of input files
2. Outputs

- List all outputs that the \( \mu \)P system must produce
- For each output,
  - What is the exact form of the signal which must be produced?
  - When must it be available, and how does the peripheral written to know it is there?
  - How long must it be available?
  - How often does it change, and how does the peripheral know that it has changed?
  - Is there a sequence of outputs?
  - What should be done to sense and recover from peripheral failures
  - Format of reports?
3. Processing

- What is the basic procedure for transforming input data into output results or actions?
- What time constraints exist? (data rates, time delays, time constants of I/O devices)
- What memory constraints exist? (program size? data memory size?)
- What standard programs or look-up tables can be used? What are their constraints?
- What special cases exist, and how should the program handle them?
- How accurate do the results have to be?
- How should the program handle processing errors? (overflow, underflow)
4. Error Handling

- What errors could occur?
- Which errors are most likely? (human error, communication, mechanical, electrical)
- Which errors will not be immediately obvious to the system or the operator?
- How can the system recover from the errors with minimum loss of data or time?
- Which errors or malfunctions cause the same system behaviour? How can these errors be differentiated?
- What errors can occur on power-up or power-down?
5. Human Factors

- Critical to product acceptance and success
- What input procedures are most natural to the operator?
- Can the operator easily determine how to begin, continue and end the input operation?
- How does the operator know when data has been entered correctly?
- How does the operator recover from input errors?
- What operator errors are most likely?
- How is the operator informed of procedural errors?
- Are displays easily read and understood?
- Is the system easy to use?
- Can the operator always determine the state of the system after a distraction?
Consideration of these aspects will lead to the *Requirements Document*, or Functional Specification

- Formal statement of software requirements
- Minimal specification for an acceptable product
- Negotiated with the client, and written in client (user) language. Usually forms the basis for a contract.
- Must not specify the solution or the design
- Must also specify all areas of uncertainty and possible change

... and leads to the *Software Architecture*. 
Software Architecture (High-level design)

- Overview of system organisation
- Define the major modules, including interfaces
- Flexibility to accommodate required changes
- Define major data structures and access routines
- Identify important algorithms
- Define the user interface
- I/O, including error detection
- Memory management – estimate max. memory
- String storage – e.g. error messages and prompts
- Error processing
Software Architecture

- How should the architecture be described?
  - Many ways...
  - Combination of diagrams and text
Flawcharting can be useful in the early phases of a design. That is, until about 1968...
Flawcharting

Advantages
- Standard symbols – punched card, paper tape…
- Understood by non-programmers, and used in other disciplines
- Can provide a pictorial representation of the entire system

Disadvantages
- Difficult to design, draw or change. Becomes messy.
- Flawchart shows only the program organisation and cannot show the data structure or organisation of I/O modules
- Do not help to identify or locate hardware or timing problem
- Encourages undisciplined design, with lots of jumps back
- Cannot be executed or debugged…
State-Transition Diagrams

- We require an *engineering description* of how a machine works – often, only the programmers know exactly what the code does.

- *State-Transition Diagram* provides a complete description of the machine operation, *independently* of the programming language used.

- Technique borrowed from sequential logic designers

- Common notation that can be understood by all of the design / management team

- Invaluable technique for the design, development and documentation of real-time software
State-Transition Diagram Definitions

- **State**: A period in which the system is performing a “well-defined” operation. The definition of a system’s states is often subjective.
  - States are depicted as circles, each marked with a succinct description of the state.
  - Each state can have a number of **entry** and **exit** points (transitions).
  - In a *Hierarchical State Diagram*, each state can have a state-transition diagram within it ⇒ different layers of complexity.
  - Each state is often expanded to a one-page written description using (for example), pseudo code.
State-Transition Diagram Definitions

- \textit{Transition} between states occurs when one or more variables change, due to
  - Changes to inputs (events)
  - Passage of time
  - Changes to internal values

- Transitions are depicted as curved arrows

- Each transition arrow is marked with
  - Conditions that trigger a transition
  - Actions taking place during the transition
STD Example: Battery Charger
STD Code Generation Framework

- Each state has an
  - Entry function
  - Action function
  - Transition test function(s)

- Entry function
  - Executed on initial entry to the state
  - Does any setup necessary
  - Initializes variable values

- Action function
  - Executed on each scan of the STD, as long as system stays in the state

- Transition test function(s)
  - Each function tests for one transition
  - A function returns TRUE for transition
  - If no test is TRUE, stay in the same state
Then, to implement the STD in code:
- Repeatedly scan the STD, making one pass through the set of functions
- Test the functions in the order of priority
- First function to be TRUE causes transition, and the others are not tested.
- The functions must be nonblocking: allow entry, action and transition functions to be scanned once.

Worst-case latency (scan time) is based on the slowest function
- On entry, entry and action function run
- All test functions run if there is no transition.

Can use the C `switch` construct to build transition logic
Data Flow Diagram

- Shows the flow and transformation of data in its various forms – both hardware & software

- Rectangles represent operations that transform data

- Arrows represent data input or output to or from transformation

- Each arrow is annotated with
  - Name and format of data item
  - Range of valid values of the variable
DFD Example: Battery Charger

- Battery voltage: 0-150 V dc
  - Voltage scaling
  - Analog-to-digital conversion
  - Binary integer
  - Scale conversion to engineering units

- Battery voltage: 0-10 V dc
  - Check battery temperature against upper and lower limits
  - To remainder of system

- Battery current: 0-75 A dc
  - Current-to-voltage conversion
  - Analog-to-digital conversion
  - Binary integer
  - Scale conversion to engineering units

- Battery current: 0-10 V dc
  - Battery voltage: (floating point)
  - Compare battery voltage to gassing voltage
  - To remainder of system

- Battery temperature: -20° - +40°C
  - Temperature-to-voltage conversion
  - Analog-to-digital conversion
  - Binary integer
  - Scale conversion and linearization to engineering units

- Battery temperature: (floating point)
  - Calculate gassing voltage
  - Battery gassing voltage: (floating point)
  - Battery voltage: (floating point)
  - Calculate charging current error
  - Current error
  - Apply gain, compensation and scaling
  - Current setpoint (10-bit binary integer)
  - Digital-to-analog conversion
  - Digital setpoint (0-10 V)

- Current command
  - Charging current command from other portion of controller

- Battery temperature status, three values:
  - Over temperature
  - Within limits
  - Under temperature

- Battery voltage status, two values:
  - Greater than battery voltage
  - Less than or equal to battery voltage
Program Design

General principles of all methodologies

▪ Proceed in small steps – get it working and keep it working
▪ Divide large jobs into small, logically separate tasks (independent sub-tasks can be tested and changed without affecting the others).
▪ Keep the flow of control simple
▪ Emphasise clarity and simplicity at first – you can improve performance when the system works!
▪ Be thorough and systematic: use checklists and standard procedures
Program Design

General principles (continued)

- Use simple, consistent terminology – do not worry about repetitiveness
- *Do not* start coding until the design is completely formulated
- The system must be tested, debugged and maintained – plan for this during design
- Beware of factors (specifications) that may change. Make implementation of likely changes as easy as possible by isolating them
Design Methodologies

- Modular design
- Structured programming
- Top-down implementation
- PDL-to-Code Process

These techniques represent some approaches to rational, unified design. There is no "correct" way to design every program.
Modular Design

Idea: split the entire program into sub-tasks, or modules

- Each module is a collection of functions (methods) and data which allow a number of related tasks to be performed.

- A module will have “public” and “private” functions and private data. Private entities are hidden within the module. Public entities may be accessed by other modules.

- Hide all the secrets that no other module needs to know inside the black box.

- Increase the level of abstraction.

- Often data is exported only through access routines.
Modular Design

- A module can be thought of as a software object that contains a collection of some related data and functions (methods) that operate on the data of the object
- A module may be entirely composed of software
- A module can conceptually contain some hardware elements (sensors or actuators). In this respect it provides a software representation of the physical devices together with methods to command or interrogate their states
  - Modules of this type are said to provide a “hardware abstraction layer” (HAL)
Guidelines for Modularization

- Modules should be as distinct and logically separate as possible
- Modules should share only essential information
- Modules that reference common data should be the same
- Use access routines to operate on data
- Modularise to hide:
  - Areas likely to change – hardware interface, input/output formats, non-standard language features, things that were hard to implement, business rules – e.g. tax laws
  - Big data or complex data – use access routines
  - Complicated logic or algorithms
  - Programming language constructs – work at one level of abstraction higher
Modules (Objects) in C

Public Entities
- Public Functions are
  - *Declared* in file *module.h* (public “header file”)
  - *Defined* in file *module.c*
- Public Data *does not exist* – all data are private

Private Entities
- All private Functions and Data are
  - *Declared* in file *module.c*, *not* in *module.h*
  - *Defined* in file *module.c*
- Private Functions
  - Cannot be called by functions defined outside of *module.c*
  - Return *static* and are therefore not visible to the linker
- Private Data
  - All data is private
  - All data *must* be accessed using (public) access functions
Modules (Objects) in C

Distribution
- Public interface distributed in module.h
- Everything else distributed in module.lib

- See examples in Coursework Files on Server
  main.c and main.h
  commands.c and commands.h
Functional Decomposition

- As well as the STD and the data flow diagram, make a list of all functions required within the system
- Begin from the system’s functional specification
- Use top-down design
Functional Decomposition

- Example: preliminary decomposition of battery charger

![Diagram of battery charger functional decomposition]
- Use stepwise refinement until modules “appear”.
- A module
  - Can be described unambiguously on one page
  - Performs a group of strongly related functions on strongly-related data
  - Appears as a “black box” to other modules (information hiding principle)
  - Requires rigorous definition of the public interface to remainder of system. Minimum data passing
A tree diagram can clarify the hierarchical structure of the system.

Seek to reuse functions where possible.
Example: one functional decomposition of measure temperature
PDL – Program Design Language (Pseudocode)

- English-like statements that describe specific operations
- Programming language independent, but indented to show structure and control flow.
- Written at the level of *intent*, not implementation
- Expanded (refined) successively in more and more detail until the code “appears”.
PDL-to-Code Process

- Precursor: Know the Module’s function
- Design the Routine
  - Check the Requirements
  - Check the Architecture
  - *Name* the routine
  - Consider and decide on testing
  - Investigate appropriate algorithms and data structures
  - Write the PDL
- Code the Routine Stub Interface
  - Turn the PDL into comments in your language-of-choice
  - Code the first (argument access) and last (return) statements

⇒ a stub *that can be compiled and executed*
PDL-to-Code Process

- Expand the stub code
  - Write code to implement each comment in the stub

- Check the logic of the code informally, and clean up:
  - Interface
  - Data – names, unused or uninitialised, etc
  - Control flow – initialisation, first pass, last pass
  - Formatting – brackets, indentation, etc
  - Code documentation, comments

- Iterate back to the PDL as required
PDL-to-Code Process

- Check the logic of the code formally
  - Compile - the compiler is your friend!!
    - Highest warning level
    - Eliminate (or really understand) all compiler errors and warnings

- Validate the code
  - Single-step with debugger
  - Run the test cases
  - Remove errors and retest…
PDL-to-Code Example

PDL

// Calculate factorial of an integer

// validate maximum argument

// result = 1

// while f > 1
   // result = result * f
   // decrement f

// return result
PDL-to-Code Example

+ function signature

// Calculate factorial of an integer
unsigned long factorial( unsigned f )

    // validate maximum argument

    // result = 1

    // while f > 1
        // result = result * f
        // decrement f

    // return result
PDL-to-Code Example

+ return value = executable stub!!

```c
// Calculate factorial of an integer
unsigned long factorial( unsigned f )
{
    // validate maximum argument

    // result = 1

    // while f > 1
    //    // result = result * f
    //    // decrement f

    // return result
    return 24UL;
}
```
Structured Programming

Basic Idea

- Each program (module, object) consists of elements chosen from a limited set of control structures
- Each structure has a single entry and single exit
- Formatting of structures is usually imposed
Structured Programming: Structures

Only three basic structures required

Only three basic structures permitted:
1. Sequential Structure
   \[ S_1 \]
   \[ S_2 \quad \text{where} \quad S_i \quad \text{is a statement or a structure} \]
   \[ S_3 \]
2. if-then-else Structure
   \[ \begin{align*}
   & \text{if } C_1 \\
   & \quad \text{then } S_1 \\
   & \quad \text{else } S_2
   \end{align*} \]
3. Loop Structure
   a) while \( C \) do \( S \) – zero or more loops
   b) do \( S \) until \( C \) – one or more loops

Sometimes a fourth case (switch) structure is added
Structured Programming: Advantages

- Forces more discipline in the programmer, resulting in more systematic and better-organised programs
- Simple sequences of operations – easy to
  - Test
  - Debug
  - Maintain
- Consistent with modular programming
- Partially self-documenting
- Easy to describe with PDL
- Increases programming productivity

– but –
Structured Programming: Disadvantages

- Only some languages (C, C++, Ada, Modula, ...) directly support the structures
  - Forget those that don’t!!
  - Note: Assembly language does not support structures, but see the structured assembler preprocessor in Peatman
- Structured programs often execute slowly
- Having only three structures can make some logic difficult to construct, leading to confusing code
Structured Programming: General Rules

- Begin with pseudocode (PDL)
- Use *sequential*, *if-then-else* and *do-while* structures – they are a complete set
- Indent each level, and use terminating braces
  ```
  if ( )  
  { 
  } 
  if (year >= 2000) 
  { 
  } 
  y2k = TRUE;
  ```
- Emphasise simplicity and readability
- Comment and document the program
- Check the logic – first and last, typical, special cases
Top-Down Implementation – Not

Bottom-Up Implementation

- Write all the modules (objects)
- Test each separately
- Get them to work together to form a complete program

– hmmmm – I can see some rocks ahead…
Top-Down Implementation

How can we test and debug modules (objects) in the actual programming environment without wasting effort rewriting code?

- Write the top-level supervisory program, `main()` — this may just be a sequence of function calls
- In place of (undefined) functions, use function stubs
  - Access the function arguments
  - Return a legal value
- Test the supervisor function, `main()`
- Expand the stubs, representing internal calls with more stubs…
Top-Down Implementation

- An “executable prototype” of our project exists at all times during project development
- Functionality is added gradually
- Testing occurs continuously – project always runs

- Top-down implementation requires modular (or object-oriented) programming, and is compatible with structured programming and the PDL-to-code process
Top-Down Implementation: Advantages

- **Greatly** increased productivity in large projects
- Testing and integration occurs at each level of expansion
  - *Unit Testing* is testing of a module (object) in isolation
  - *Integration Testing* is testing of a module (object) in the project
- No special driver programs are required for testing
- Applicable to all kinds of design and planning tasks
Top-Down Implementation: Disadvantages

Not Many!

- The design may not mesh well with system hardware
  - So do bottom-up implementation at HAL
- May not encourage software reuse
  - May not be able to use existing software
  - May not lead to modules of general applicability

- Good OOD will often overcome these two objections
Incremental Implementation

- Use "iterative development" (spiral development model)
  - The repeated cycle of
    Analysis -> Design -> Code -> Test
    produces a series of prototypes of increasing capability

- Define the minimal system for a first build
- Once the first build is achieved, the system always "works"
- Successive builds integrate new or expanded (tested) code modules to add functionality
- Early risk reduction, high priority features build first, early availability, etc...
General Design Guidelines

- Begin with a STD to define the architecture
  - Discrete states
  - Flow of control
- Identify necessary modules (objects)
  - Public functions
  - Private functions
  - Data structures
- Watch for common tasks or data structures
- Write PDL of `main()` and modules (objects) to define
  - Function interfaces
  - Data instantiation
- Use structured programming
- Emphasise simplicity and readability always
General Design Guidelines

- Use (and enforce) a coding standard
- Use PDL-to-code process
  - Make the stubs a complete and separate as possible
  - Define precisely all outcomes from each stub
  - Expand each stub one level at a time

- Test and debug
  - Unit tests before each expansion
  - Integration test after each expansion

- Be aware of hardware constraints
  - Don’t be afraid to do some bottom-up design when necessary
Verification (Testing)

It’s not correct until demonstrated to be correct…
Verification (Testing)

- Plan for testing during system design
- Similar to debugging
  - Typical cases
  - Special cases
  - Trivial cases

- Testing should be
  - Modular
  - Structured
  - Top-down
Verification (Testing)

- **Unit Testing**
  - Is each unit (function, module) correct and fit for use?
  - Requires a set of known input data and known results

- **Integration Testing**
  - Takes known-good (unit tested) modules combined into an integrated system and verifies correctness
  - Requires a set of integration tests with known input data and known results
Unit Testing

- Unit: A single individual or thing regarded as a member of a group or number of things or individuals, or discriminated from these as having a separate existence; one of the separate parts or members of which a complex whole or aggregate is composed or into which it may be analysed. [OED]

- The purpose of a testing a unit is to ensure that it meets all functional and technical requirements. The design of a set of unit tests should be complete and also punish over-inflated units that can be broken down further.
Unit Testing

- While a unit can only be tested during / after its development, designing the unit test can help with determining the breakdown of a project into units.
- Technical requirements: Are the software / hardware outputs within technical requirements – e.g. voltage output.
- Functional requirements: Does the output of the unit make functional sense; are the values ‘sensible’?
- Errors: How are they handled? How might they occur? How can we catch them?
- This list is not exhaustive...
Integration Testing

- Once units have been tested individually, it is then important to test that units work with ‘associated’ units.
- Also needs a well defined list of tests to pass.
- Proper breakdown of a project with good stub generation, documentation and DFD diagrams etc. should make this testing painless.
- Ideally units should be tested together in a way that allows for the cause of errors to be pinpointed, e.g. test units A+B, then B+C before testing A+B+C.
- As an example, testing from ‘bottom-up’ can be a methodological way to test units together.
Revision Control

- A revision control system is a repository of project related files, often source code, but can also include documentation and other files. Changes are tracked with why the change was made, by whom and when.
- Contain the ability to reverse changes when necessary and is a vital part of good programming discipline.
- Often good to make lots of small changes to a project so if a rollback is required there is good resolution available.
- Examples include Subversion (SVN) and GIT.
Debugging Tools & Techniques
Debugging

Tools useful during debugging (and testing) include

- Simulator
- Oscilloscope
- Logic Probe
- Logic Pulser
- Logic analyser
- In-circuit debugger
- In-circuit emulator
Simulator

- Computer program that simulates the operating cycle and internal state of another computer, keeping track of contents of registers and memory
- Helps with the software side of debugging
  - Single step through code
  - Set break points
  - Inspect (and change) contents of registers and memory
- Cannot help with hardware, I/O and timing
  - Does not (usually) run in real time – can be slower or faster
  - Cannot fully represent I/O
Monitor

- One software part resident in the target processor
- Another software part in the host – often a PC
- Communicate by serial cable or USB

Typically allows:
  - Control of target processor from host
  - Load and dump code to/from target
  - Set breakpoints and single-step (often only at assembly language level)
  - Examine and change contents of registers and memory
Oscilloscope

- Learn to use it *really well*!

- When debugging hardware – e.g. signal levels and relative timing – write very simple *assembly language* fragments to exercise the hardware

- Assists with isolating problems, triggering
Logic Analyser

- Basically a digital oscilloscope with 32, 64, or more input channels
- Flexible and powerful triggering – for example, the combination of addresses in a particular range and a particular data sequence
- Can help find
  - Timing or signalling errors
  - Incorrect signal sequences
  - Glitches
In-circuit Emulator

- Emulates the target processor in the target hardware
- Useful for debugging target hardware

Typically allows, at full target clock speed
- Control of emulation processor from host
- Load and dump code to/from emulator
- Set breakpoints and single-step
- Examine and change contents of emulation registers and memory
Debugging Checklist

- Many errors can be found by simple hand-checking, either on paper or with a debugger
- Adopt systematic approaches to fault finding
- Compare STD, PDL with the actual code
- Examine loops
  - Everything initialised correctly
  - First and last passes correct?
  - Does zero loops work?
- Examine branches
  - Are the tests consistent with the STD?
  - Check each branch with 2 simple cases
Common Programming Errors

Many apply to HLL as well as assembly language, but are harder to make in HLL

- Forgetting to initialise (or reinitialise) variables such as counters, pointers, sums, etc.
- Inverting the logic of conditional jumps
- Updating counters or pointers in the wrong place, or not at all
- Program failure in trivial cases, such as no data
- Reversing the order of operands
- Inadvertently changing flags before you test them
- Confusing data and addresses
- Accidentally reinitialising a pointer or memory location
Common Programming Errors

- Confusing binary and BCD numbers or decimal and hexadecimal numbers
- Reversing the order in subtractions
- Ignoring side-effects of assembler functions (e.g. flag bits)
- Using shift instructions incorrectly
- Counting lengths incorrectly – e.g. 4 locations in 0x100 – 0x103
- Forgetting that 16-bit numbers occupy two successive memory locations
- Forgetting to initialise the stack pointer
- Confusing the stack pointer and the top of stack location
- Changing a register or memory location before using it
Common Errors in Interrupt-Driven Code

- Confusing edge- and level-sensitive interrupt pins
- Forgetting to re-enable interrupts after servicing an interrupt request
- Forgetting to save and restore required registers or memory locations
- Restoring from the stack in the wrong (reverse) order
- Leaving results in registers and overwriting them during restoration
- Re-enabling interrupts before the necessary conditions are established
- Not installing handlers for all interrupts, even those that you don’t think you are using
Documentation

- Good documentation is crucial in projects that involve assembly language programming
- Documentation helps the programmer during code debugging and verification
- Essential for later use – maintenance or extension of programs
Documentation Could Include

- Written description of the program
- State-Transition Diagrams
- Data Flow Diagrams
- Flawcharts
- PDL Outlines
- Parameter and definition lists (assembler)
- Annotated program listing
- Memory Map
- Test Plan and test results
- Software Maintenance Manual
- User Manual (for engineer)

- Written description of the hardware
- Circuit Diagrams
- Start-up and Shut-down Procedures
- Adjustment and Calibration Procedures
- Mechanical Drawings
- Hardware Maintenance Manual (for Engineer)
- User Guide (for consumer)
Self-Documenting Programs

- Other techniques cannot substitute for self-documentation – code readable without comments
- Some guidelines for self-documenting programs
  - Use meaningful names and labels
  - Use “verbs” for function names, “nouns” for data names
  - Choose the obvious name – don’t be clever!
  - Use full words for names where possible – don’t shorten
  - Keep names as distinct as possible
  - Use clear, simple structures with as few jumps as possible
  - Don’t worry about execution speed, memory use – concentrate on simplicity
Commenting Guidelines

- Don’t repeat the code – explain at the *level of intent*
- Never comment the obvious – it just obscures the useful comments
- Use standard terms – don’t worry about repetition
- Comment each obscure or important point
- Revise obscure comments
Commenting Guidelines

- Fix the code ⇒ fix the comment (add revision note)
- Use standardised keywords as development notes – for example: TODO, BUG, HACK, REV
- Place comments at the start of a sequence or on the line to which they belong
- Use file header comments and function comments – see the files module.c, module.h for examples
Use *Doxygen*!

- *Doxygen* is a general purpose *JavaDoc*-like documentation generator
- Generates documentation from special comments in the source code
- Hypertext output in HTML, PDF, RTF formats, etc.
- Basic *Doxygen* templates in ConTEXT – use CTRL-J
- See [http://www.doxygen.org/](http://www.doxygen.org/)