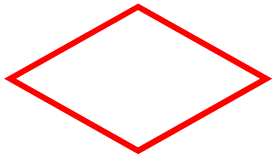


Disciplined Software Development

Lecture Notes 9

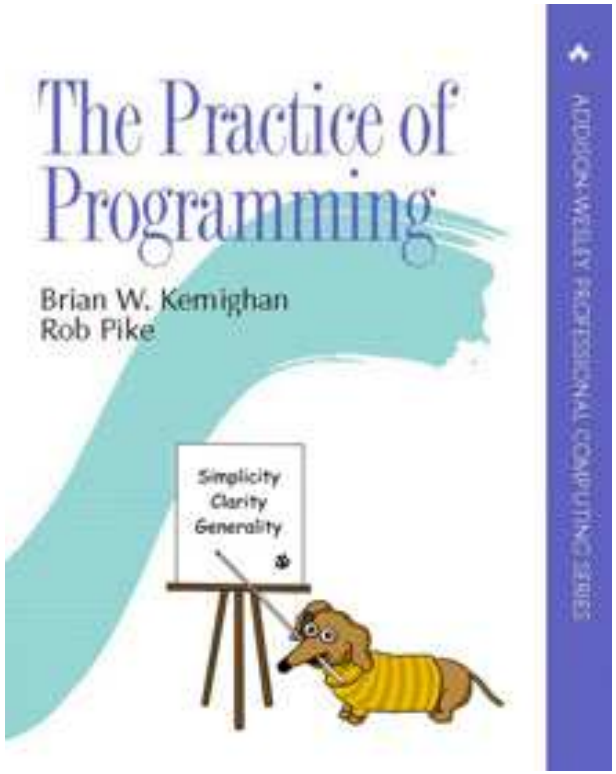
Overview

Software Goals

Why *design* software before *coding* it?

How should software be designed?

How should software be coded (written)?



Useful book (explains guidelines and much, much more)

- **The Practice of Programming**, Brian W. Kernighan & Rob Pike, Addison Wesley 1999

Software Goals

Simplicity – program is short and simple

Clarity – program is easy for humans and machines to understand

Generality – program can be used for a broad range of situations

Software Design

How do you think companies create software?

Just dive in and start writing code, or

Plan the architecture and structure of the software?

Software is like any engineering project - you need to identify WHAT you want to do and HOW you want to get there.

WHAT = requirements

HOW = development process

How do you know you developed the software successfully?

Compare the finished product to the requirements (compliance)

Why Bother?

"He who fails to plan, plans to fail"

Most companies have an established process for developing hardware and software.

Software development processes can differ between companies, or even between projects in the same company.

Software development can occur at the same time that hardware is designed (co-development), especially with embedded products. A delay in either affects the timing of the other.

What is an Algorithm?

A formula? A solution? A sequence of steps? A recipe?

A former Vice-President? (Al-Gore-ithm?)

An algorithm is created in the design phase

How is an algorithm represented?

Typically represented as pseudo code

Historically represented as flowcharts

Do yourself a favor – write algorithms before code – always!

Pseudo Code

Pseudo code is written in English to describe the functionality of a particular software module (subroutine)

Include name of module/subroutine, author, date, description of functionality of module, and actual steps

Often you can take the pseudo code and use them lines in your program as comments!

Avoid a very fine level of detail (although this may sometimes be difficult to do)

Avoid writing code – use English, not assembly language (or higher-level language) instructions

Software Design Process

Study the problem FIRST (THINK!!).

Write the important parts of your problem down.

Break the problem into manageable pieces. Solve each of the pieces individually.

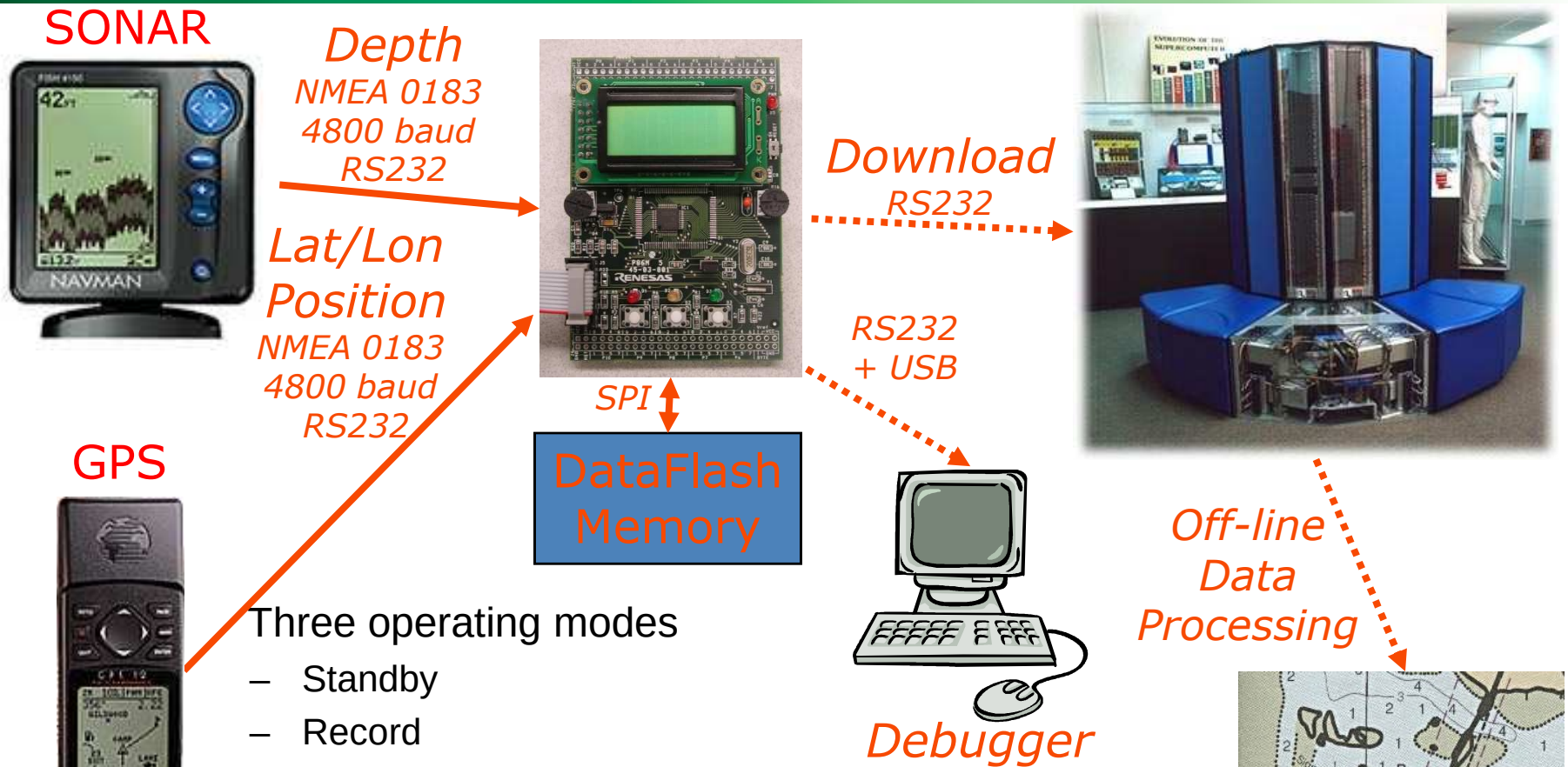
Write an algorithm of the solution of your problem, or for each piece you identified.

Create test cases for your program (more later).

Write the code. Include comments as you code.

Test your program

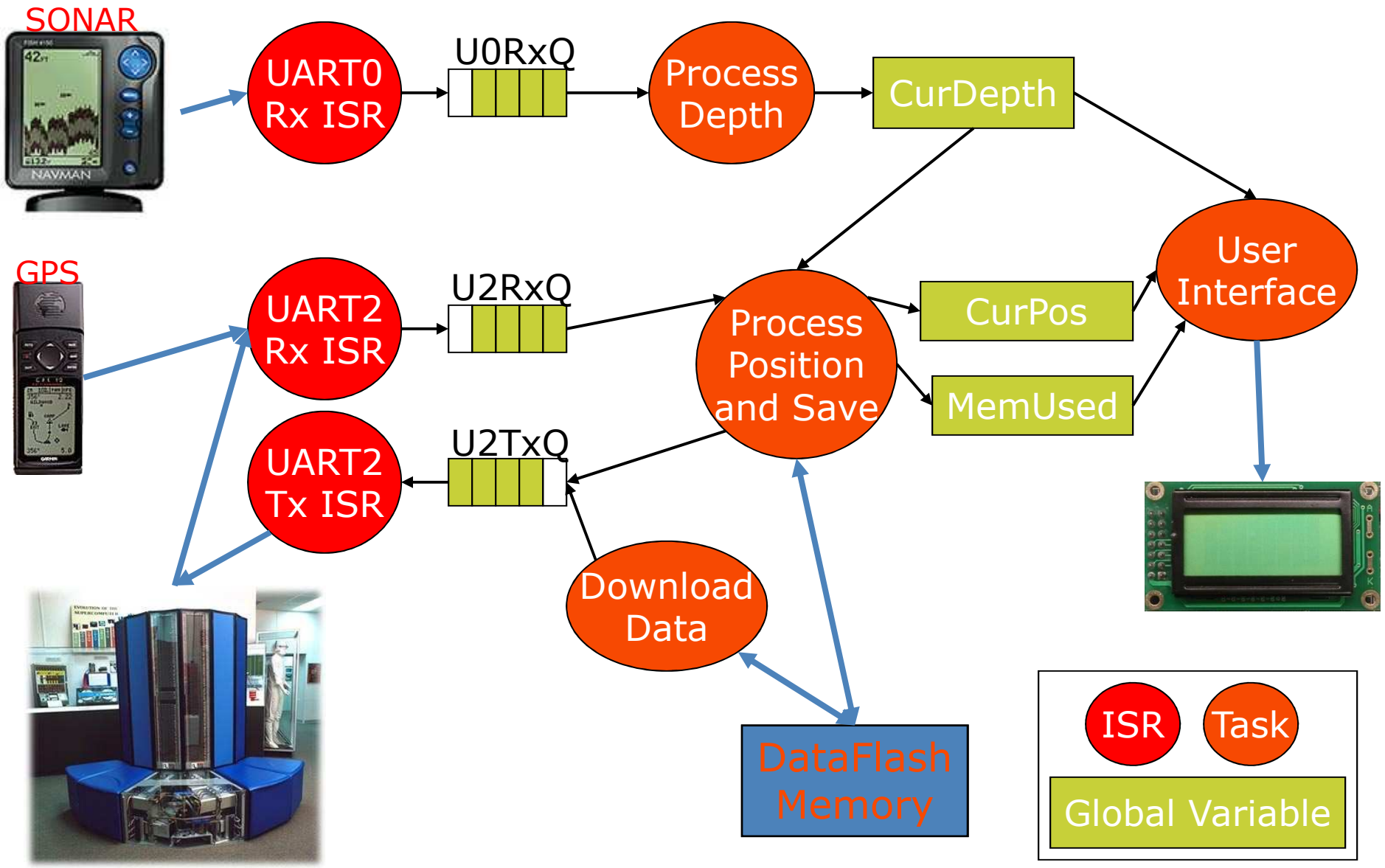
Data Logger General Requirements



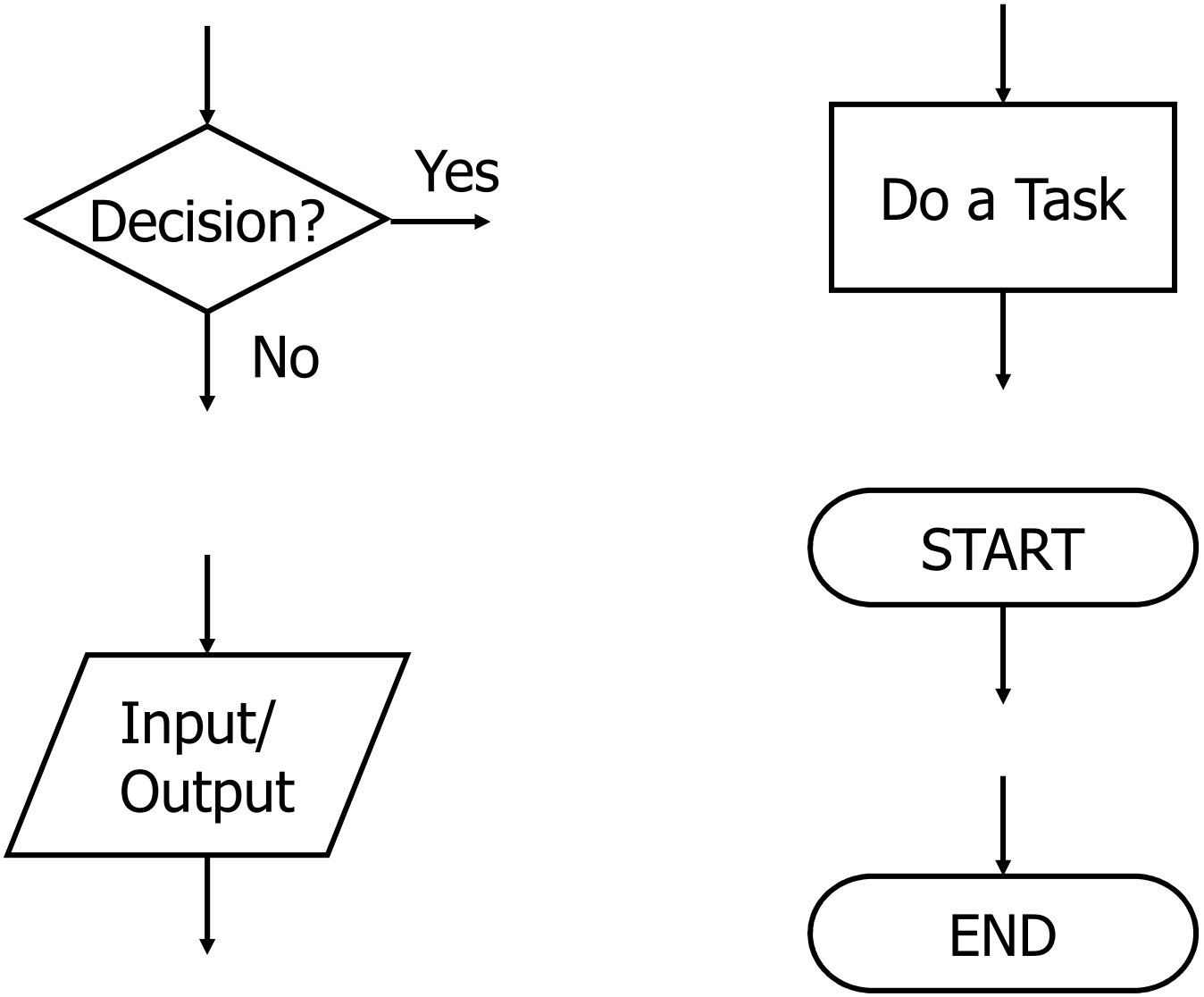
Three operating modes

- Standby
- Record
  - Get depth information from Sonar via UART0
  - Get position information from GPS via UART2
  - Store most recent depth from each position in DataFlash
- Download: Download depth and position information from DataFlash

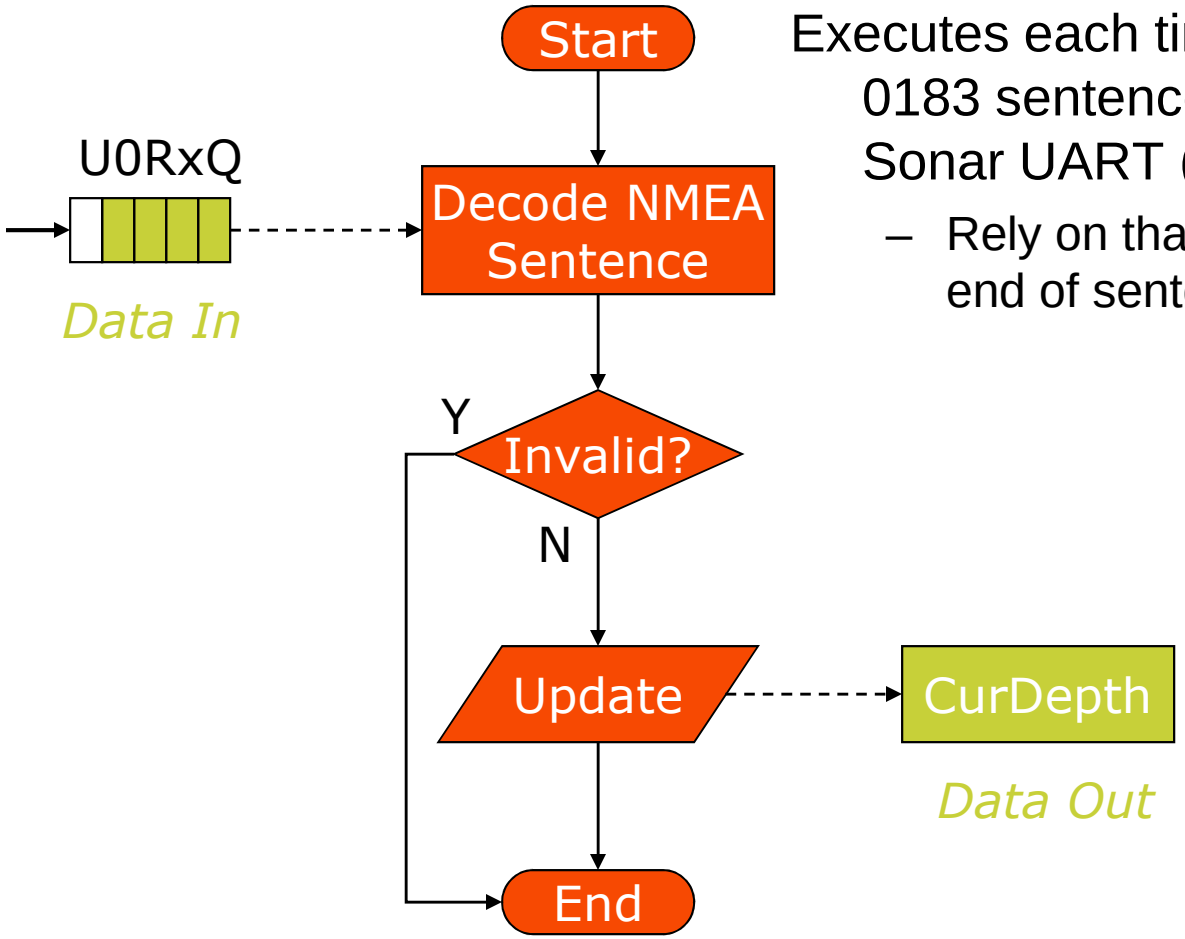
System Tasks and Data Flow



Flowchart Symbols (Control Flow)



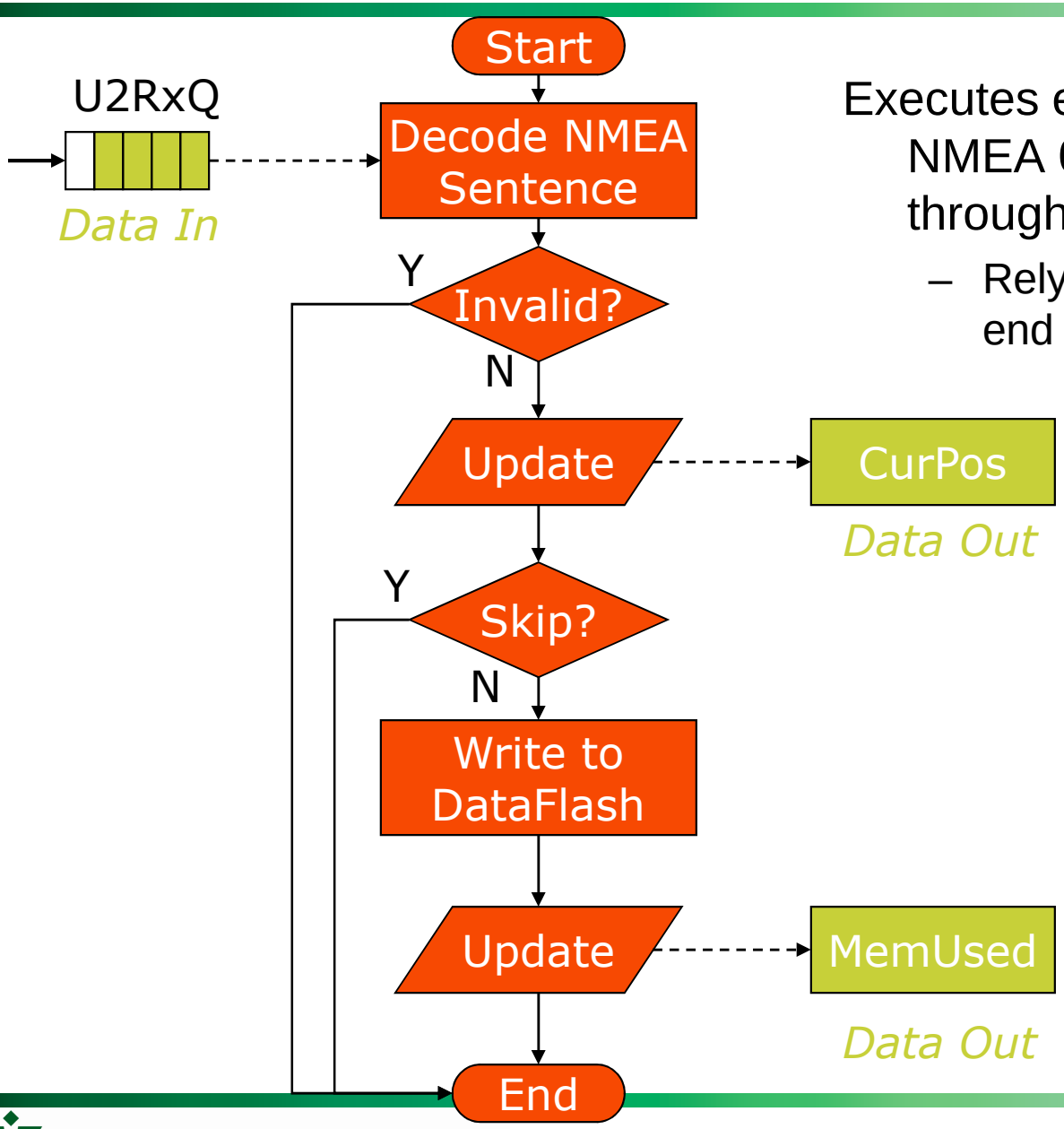
Flowchart for Process Depth



Executes each time a complete NMEA 0183 sentence arrives through Sonar UART (#0)

- Rely on that Receive ISR to detect end of sentence

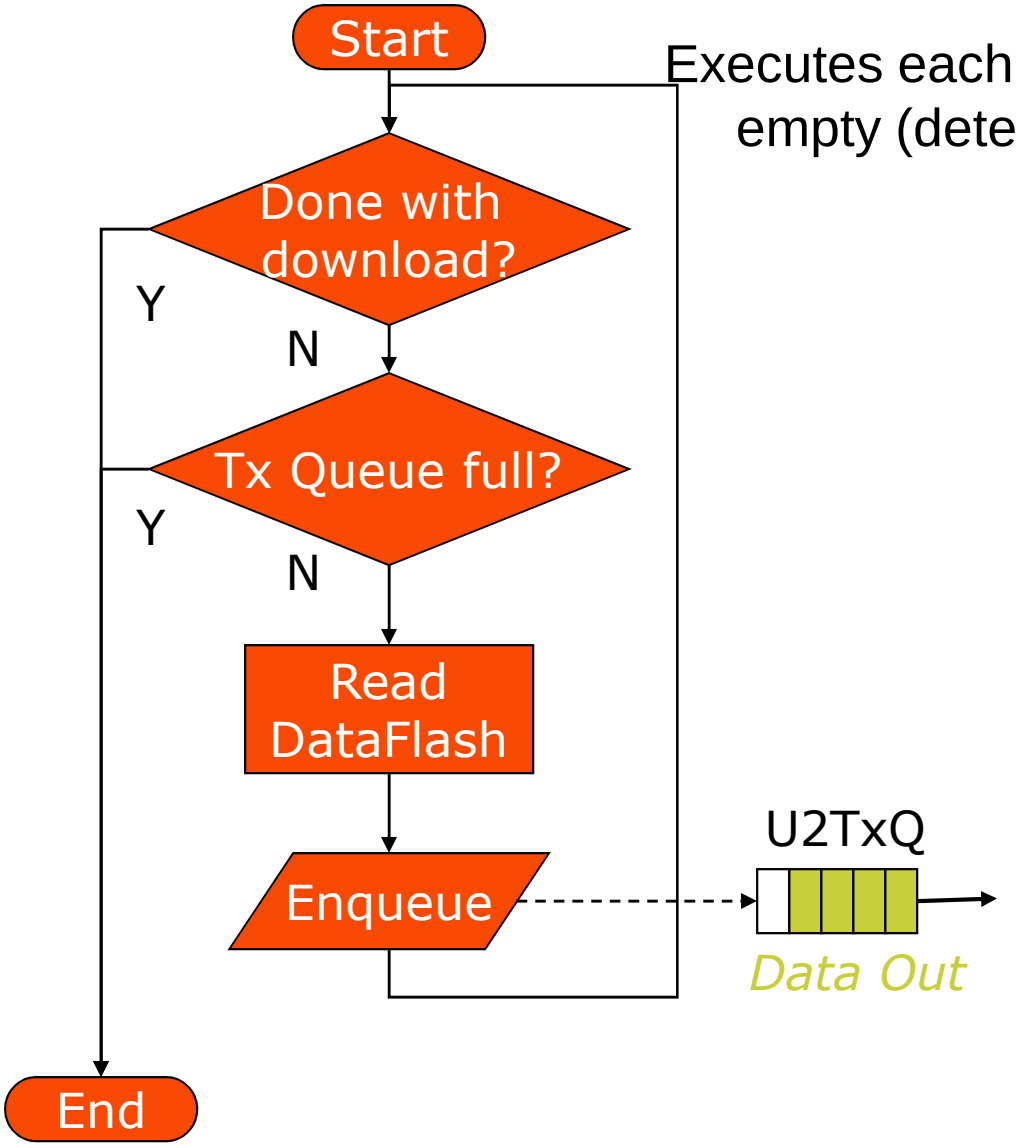
Flowchart for Process Position and Save



Executes each time a complete NMEA 0183 sentence arrives through GPS UART (#2)

- Rely on that Receive ISR to detect end of sentence

Flowchart for Download Data



Executes each time U2TxQ becomes empty (determined by U2 Tx ISR)

Coding Style Guidelines

1. Names
    1. Use descriptive names for global variables, short names for locals
    2. Use active names for functions (use verbs): Initialize_UART
    3. Be clear what a boolean return value means! Check_Battery vs. Battery_Is_Fully_Charged
2. Consistency and idioms
    1. Use consistent indentation and brace styles
    2. Use idioms (standard method of using a control structure): e.g. for loop
    3. Use else-if chains for multi-way branches

Coding Style Guidelines

3. Expressions and statements
    1. Indent to show structure
    2. Make expressions easy to understand, avoid negative tests
    3. Parenthesize to avoid ambiguity
    4. Break up complex expressions
    5. Be clear: child = (!LC&&!RC)?0:(!LC?RC:LC);
    6. Be careful with side effects: array[i++] = i++;

Coding Style Guidelines

4. Macros
    1. Parenthesize the macro body **and** arguments

       #define square(x) ((x) * (x))
5. Magic numbers
    1. Give names to magic numbers with either #define or enum

       #define MAX_TEMP (551)

       enum{ MAX_TEMP = 551, /* maximum allowed temperature */

                MIN_TEMP = 38, /* minimum allowed temperature */ };
    2. Use character constants rather than integers: if ch==65 ???? if ch ==‘A’
    3. Use language to calculate the size of an object: sizeof(mystruct)

Coding Style Guidelines

6. Comments

1. Clarify, don’t confuse
2. Don’t belabor the obvious
3. Don’t comment bad code – rewrite it instead
4. Don’t contradict the code

Coding Style Guidelines

7. Use a standard comment block at the entry of each function
    1. Function Name
    2. Author Name
    3. Date of each modification
    4. Description of what function does
    5. Description of arguments
    6. Pre-conditions
    7. Description of return value
    8. Post-conditions

Coding Style Guidelines

8. Defensive programming
    1. Upon entering a function, verify that the arguments are valid
    2. Verify intermediate results are valid
    3. Is the computed value which is about to be returned valid?
    4. Check the value returned by any function which can return an invalid value
9. Every function should be no more than 60 lines long, including comments. (Why?)

Statistics on Software Projects

Standish Group International, reported in 1995:

Only 16% of software projects were expected to finish on time and within budget

Projects completed by the largest American Organizations had only 42% of their originally proposed functions

31% of software projects were cancelled before completion costing the US economy $81 billion

NASA software research data indicates that 40% of the cost on large projects is spent on rework

Advantages of SPI Efforts

Organizations that have invested in software process improvement for 3+ years report average yearly gains of:

- 37% productivity
- 18% more defects found in pretest
- 19% reduction in time to market
- 45% reduction in field error reports
- The average return on investment is 4:1

So what is the CMM??

CMM, the Capability Maturity Model, is an engineering practice model for an evolutionary process improvement cycle. There are five levels:

Initial – unpredictable and poorly controlled (chaos)

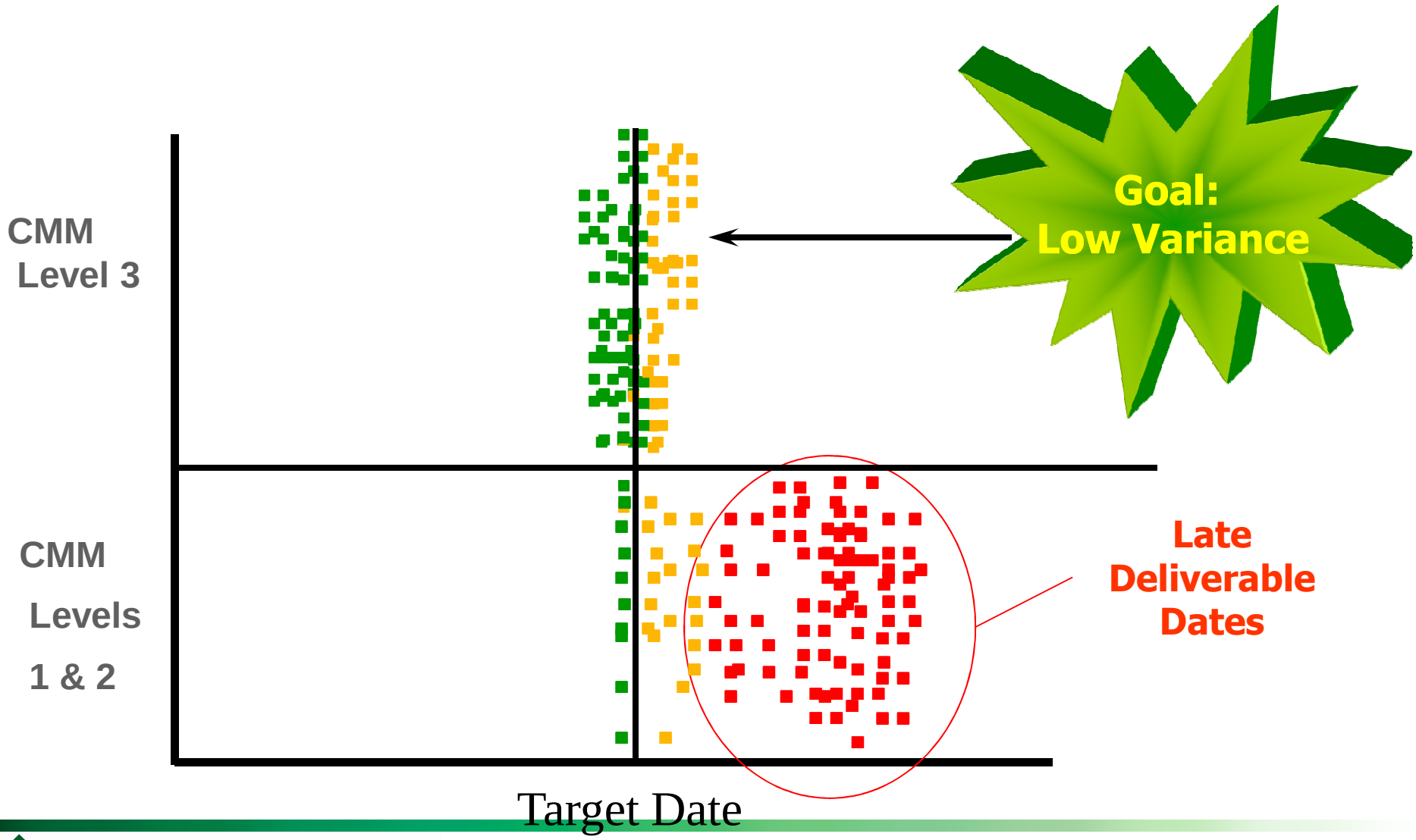
Repeatable – can repeat previously mastered tasks

Defined – Process characterized and fairly well understood

Managed – process measured and controlled

Optimizing – focus on process improvement (Space Shuttle!)

Boeing's Success



How do we mature through CMM?

We initiate the need for improvement

We diagnose our current environment

We establish our plans and action teams

We develop the solutions using our teams in concert with our plans

We leverage what we have created in the next improvement initiation

Each stage has additional tasks that need to be done, i.e. peer review, software configuration management.

Software Development Environment

Companies that develop code need to ensure they employ "Software Configuration Management" (SCM).

Basically all work products that will be delivered as part of a project are controlled by SCM through baselines that are established at the beginning of the project.

A company has a "library system" (repository) where developers check out code they will change. They check it back in when done.

Everyone has a copy of the entire code base so they can locally compile the code, adding the few changes of the code they have checked out.