Embedded Systems Engineering

System Reliability - 2

- Application software aspects
  - Basic design issues
  - Run-time problems
- Real-world interfacing
  - Identifying & detecting faults on inputs
  - Avoiding faults on outputs
- Operating systems aspects
  - Protection techniques
  - Tasking models & scheduling
  - Distributed applications

Michael Brockway

Application Software Aspects

- Basic program design issues
  - Use rigorous design techniques
  - Use well-ordered program structures
  - Develop & use good programming standards
  - Write *readable* code
  - Use an appropriate programming language. For critical applications -
    - Spark Ada, Ada 95, MISRA C
  - Use code quality checking tools
    - Lint for C
  - Do not use unconditional transfers of program control
  - Do not use recursion
  - Avoid pointers
  - Limit use of inheritance, polymorphism
  - For highly critical systems, do not use dynamic memory allocation
  - Use static & dynamic code analysis techniques

Application Software Aspects

- Dealing with run-time problems
  - Exception handling
  - Backward error recovery
  - N-version programming

- Exception Handling
  - Normal operation ceases: control transfers to an *exception handler*
  - Subsequent events are application-specific
    - determined by functional needs, response times, system criticality
  - It may be possible to put the system into a pre-determined acceptable state and then continue processing
    - *Forward error recovery*

Exception Handling

- Many languages do not have this. Possibly the neatest implementation is in Java: eg
  - The `OutputStream` constructors and the functions `writeObject`, `close` may **`throw`** an `IOException`:
  - Object-oriented: exceptions are encapsulated in objects of subclasses of class `Exception`

```
ObjectOutputStream str;
try {
  str = new ObjectOutputStream(new FileOutputStream(
    chooser.getSelectedFile().getName()));       //open file
  Enumeration e = strokes.elements(); //Iterate thru strokes,
  while (e.hasMoreElements()) {       //writing to file.
     str.writeObject(e.nextElement());
  }
  str.close();                        //close file
} catch (IOException x) {
  System.err.println("Could not open file for output");
  x.printStackTrace();
}
```

Exception Handling

- C++ has try/throw/catch too. But generally the programmer decides when an exception will be thrown

```
int getNextNum() {
    int n;
    ifstream.input("nums.txt");
    if (!input) throw "Input Failed";
    input >> n;
    input.close();
    n++;
    ofstream.output("nums.txt");
    if (!output) throw "Output Failed";
    output << n;
    output.close();
    return n;
}

void main() {
    try {
        cout << getNextNum() << endl;
    }
    catch (char * error) {
        cout << "Error: " << error << endl;
    }
//processing resumes from here
}
```

Exception Handling

- C has the **`assert`** macro -

```
double a, b, c, s, discr;
...
discr = b*b - 4*a*c;
assert(discr >= 0);
s = sqrt(discr);
printf("Solutions are %f, %f\n", (-b+s)/2/a, (-b-s))/2/a);
...
```

- If the assertion fails, the program is aborted with a diagnostic message on the console.
  - Uses the **`abort`** facility

Exception Handling

- C also has the **`signal`** facility -

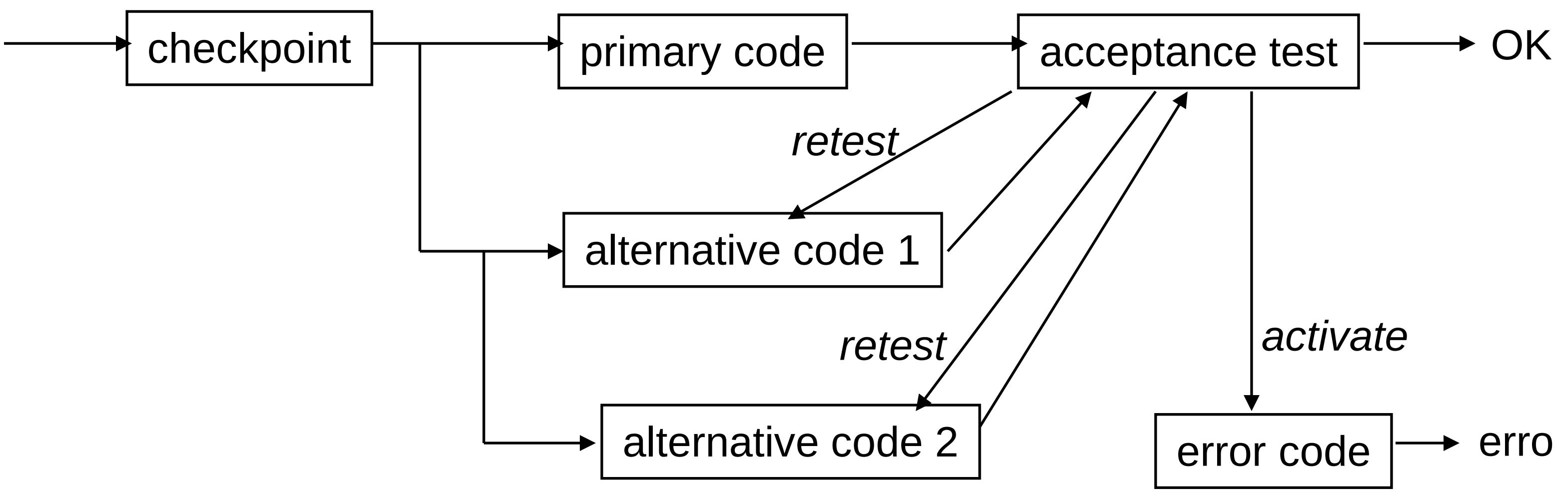
```
void myHandler(int theSignal) { ... }

int sig;
void (* oldHandler)();
/* set new handler, saving old handler */
oldHandler = signal(sig, &myHandler);
if (oldHandler == SIG_ERR)
  printf("Could not establish new handler\n");
```

- A signal can be raised by the computer's error detection mechanisms, or by a program with `raise(int sig);`
- Pre-defined values of sig which mifght be raised by the computer include
  - SIGABRT – abnormal termination (`abort` facility)
  - SIGFPE – floating point error or divide-by-0
  - SIGILL – invalid instruction
  - SIGSEGV – invalid memory access
  - SIGTERM – termination signal from a user or another program

Backward Error Recovery

- Aka *rollback*
- Maintains continuous operation when a failure occurs
- A common method is that of ***recovery blocks***:



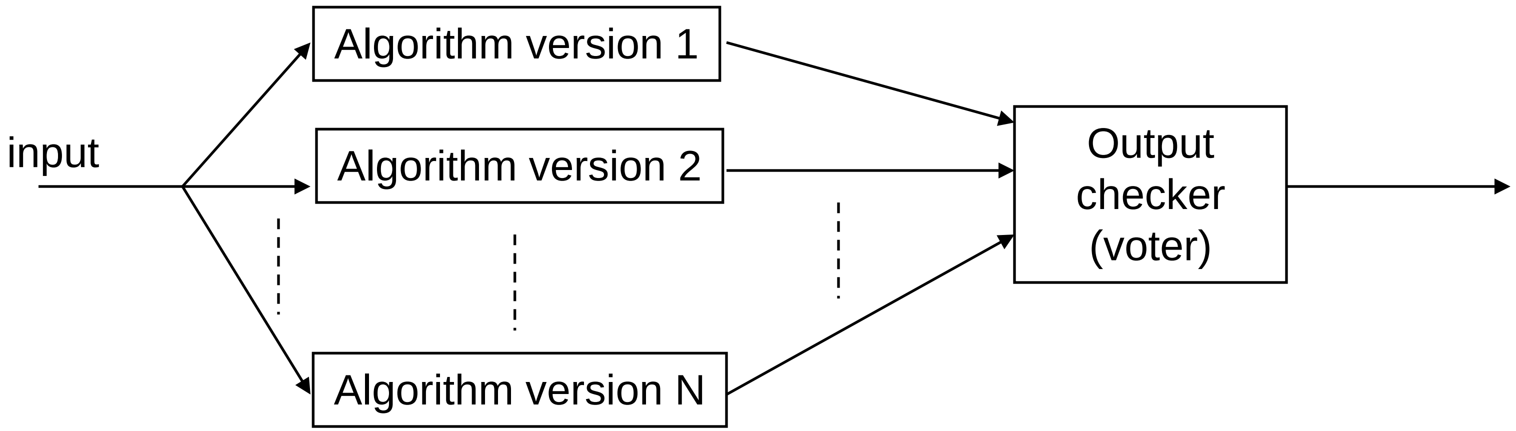
Backward Error Recovery with Recovery Blocks

- System state saved at checkpoint, then primary algorithm executed.
- If acceptance test is passed, processing continues
- Otherwise,
  - processing is *rolled back* to the state at the checkpoint
  - processing resumes with alternative code 1
  - acceptance test applied again
- On another failure,
  - roll back again
  - alternative code 2
- etc
- If all alternatives tried and acceptance test still fails,
  - execute error code

Backward Error Recovery with Recovery Blocks

- This approach is more suited to mission-critical than safety-critical systems because
  - A program failure may cause operation to stop completely until some form of external recovery is put into action;
  - Actual execution times may vary from run to run
    - performance problems
    - hard deadlines could be missed
    - processing time is indeterminate
- Designing the Acceptance tester – 3 approaches -
  - test results against pre-defined values
  - test results against predicted values
    - eg from advance knowledge of maximum rate of change
  - determine all the input values which could have produced this output: compare with checkpoint value(s)
    - can be time consuming

N-version Programming



- Checker
  - compares all results
  - If all agree, outputs the result; otherwise
  - selects the result by majority & outputs it.
    - Normally N-1 agreements at any one time

- Roots in electronic analogue control systems but can be used in single-processor applications

N-version Programming

- No need to devise acceptance tests
  - Actual results do not matter provided we have majority agreement
  - But there is time overhead

execute alg 1	result to voter	execute alg 2	result to voter	execute alg 3	result to voter	select result

time

  - Fails when common-mode errors occur
    - all versions give wrong result
    - Minimise be using as much diversity as possible between the algorithms – each devloped by a different designer

Real-world Interfacing

- Lutz, working on Voyager & Galileo spacecraft found poor understanding of interfacing requirements accounted for 44% of all logged safety-related errors -
  - Out-of-range input values
  - Non-arrival of expected inputs
  - Unexpected input arrival
  - Inconsistent code behaviour in response to input signals
  - Invalid input data time frames
  - Out-of-range arrival rates
  - Lost events
  - Excessive output signal rates
  - Not all output data used
  - Effect of input signal arrival during non-operational mode
    - start-up, off-line, shut-down

Real-world Interfacing

- Input problems –
  - switch inputs stuck in one state
  - uncommanded change of switch state
  - sensor signals going hard over to max/min
  - analogue signals locking up
  - invalid signals due to noise
  - bias on digitised signals due to stuck bits
  - sensor drift with time
- Input fault detection methods
  - Limit testing
    - Compare actual values with known practical limits.
    - Detects mainly the "hard over" type of fault
  - Rate testing
    - Compare rates of change of actual values with known practical maxima.
    - Extracting the rate of change takes time
    - differentiation magnifies effects of noise
    - subject to "false alarm" errors

Real-world Interfacing

- Input fault detection methods (ctd)
  - Predicted values
    - Noise can be a problem
    - A mathematical model of the system can be used to provide the predicted values
  - Redundant inputs
    - multiple versions of inputs cross-checked
    - In critical systems, dual redundancy common; sometimes triple, quad.
    - Disagreement does not identify the fault
      - but suitable in a system which can revert to a manual mode
      - Where continuous operation needed, use majority voting
  - Time-related values
    - Ignore signals outside specific times
    - When synchronizing operations, say, over a LAN, time-stamping messages is critical.

Real-world Interfacing

- Input fault detection methods (ctd)
  - Inferred values
    - An error on an input can be inferred from values of other inputs: eg an air-conditioning unit with sensors for room temperature, unit inlet and unit outlet temperature:
      - room = 20^o, inlet = 0^o, outlet = 10^o
      - the inlet temperature sensor is the culprit
    - Can reduce redundancy of inputs needed in safety-critical systems
  - Estimated values
    - similar to model-based prediction; but a model may not be known in advance
    - Instead, the model is built "on the fly"

Real-world Interfacing

- Avoiding Faults on Outputs
  - Do not use individual bits within a word to operate separate on/off controls. Use a specific, unique word for each control
  - If each bit of a single word does operate an individual control, then
    - Store the word in memory
      - You can read it and see what state the outputs are *supposed* to be in
    - When setting outputs, write to the control word, then output this
    - Use AND, OR bit masking to ensure you do not affect other controls
  - Use state, sequence information to limit the number of operations that may be invoked
  - Use multiple signalling where operating a control incorrectly could be dire
    - should be tied into hardware so that proper status signals are obtained
  - Apply rate-limiting to analogue signals

Operating Systems Aspects

- Problems include
  - Application software interfering with the OS
  - Tasks/applications interfering with other tasks/applications
  - Unexpected functional behaviour of tasks/applications
  - Unexpected timing behaviour of tasks/applications
  - Design weaknesses in OS

- Dealing with interference – Protection techniques
  - Memory protection, to prevent writing to the data or code area of other tasks
  - Intertask communication, signalling to implement synchronisation or mutual exclusion
    - binary or counting semaphores
    - event flags
    - messages

Tasking Models and Scheduling

- Additional requirements for safety-critical systems
  - functionality is fully predictable
  - timing is fully predictable
  - guaranteed detection of disk, OS failures
  - require a small (& predictable) amount of memory
- At the *catastrophic* severity level, the following rules are a guide:
  - static task schedule (no dynamic creation/deletion of tasks)
  - All code can be statically analysed
  - Tasks run to completion, without pre-emption
  - Watchdog techniques ensure run-time bounds are not transgressed

Tasking Models and Scheduling

- At the *critical* severity level, the following rules are a guide:
  - Both periodic and aperiodic tasks are allowed
  - Fixed-priority tasks are supported; dynamic adjustment of priorities is forbidden
  - Non-pre-emptive, co-operative and pre-emptive scheduling are allowed
  - Schedule is static – no dynamic task creation/deletion
  - Periodic tasks are structured as infinite loops
  - Worst-case execution times are deterministic
  - Memory for kernel components – stacks, TCBs, ... – is allocated statically
  - All code can be statically analysed
  - No priority inversion
  - Use scheduling algorithms which lend themselves to schedulability analysis
  - Watchdogs for run-time bounds

Distributed Applications

- The system comprises a number of computing nodes linked by a network.
- Each node runs autonomously, but co-operates with other nodes by passing messages through the network
- A good design features
  - Composability
    - A large system is built by integrating a set of well-specified and tested subsystems, whose essential properties (timeliness, stability, …) are preserved by the system integration.
  - Scalability
    - The system can grow by the addition of nodes, within the capacity of the communication system, or by the replacement of a node by a gateway to another communication network supporting additional node.
    - In this architecture, complexity is kept with reasonable bounds by encapsulation.

Distributed Applications

- a good solution when the application -
  - controls physical devices that are physically dispersed
    - plant control systems
  - controls physical devices that are dynamically interchangeable or configurable
    - train control systems
  - needs to be highly reliable and fault-tolerant
    - This can be provided by providing redundancy -- replicating components.
    - In a good design, an error is localised within a relatively simple local subsystem

Distributed Applications

- Possible problems for the developer -
  - The system depends on an effective communication network. This might mean there is a single point of failure
  - Performance overhead
    - A local procedure call might take 10 to 100 μsec (microseconds)
    - A remote procedure call might take 10 or 20 msec (milliseconds) -- 200 to 1000 times as long.
  - In a *hard* real-time system, a result must be delivered within a specified time frame; otherwise the system has failed (ABS braking system, nuclear reactor control system). How does the developer *predict* real-time performance without being unduly pessimistic, when communication through a network is involved?

Distributed Applications

- Possible problems for the developer –
  - A failure could be
    - in the network
    - in a node CPU
    - in a task at a node

  - Data *addressing* errors (wrong senders/receivers)
  - Data errors (corruption)
  - Message timing errors
  - Message sequence errors
  - Consistency of data across the system
  - Synchronisation of operations across the system