CSE 237B Fall 2009

Programming for Embedded RT Systems

Programming in the large

Rajesh Gupta
University of California, San Diego
Tentative Outline of Topics Covered

Four parts:

• Programming in the large
  – Modularity, Abstraction, Reuse, Fault tolerance, Correctness, Validation

• Programming support
  – for time, space

• Programming in the small
  – Estimation and optimization

• Beyond programming
  – Control systems
  – Media processing

1. Modeling and Specification for Embedded Systems
2. Programming Embedded RT Systems
3. Fault Tolerance and Recovery
4. Exceptions and Exception Handling
5. Low Level Programming
6. Programming Language Support for Time
7. Programming Language Support for Space
8. Scheduling for Real-time, Task Management, RTOS Issues
9. Virtualization, Security
10. Optimization of Embedded Software
ES Characteristics

► Complexity in function (and in size)

• High reliability and safety
  – failure has severe life, environmental and economic consequences

• Guaranteed response times
  – predictability is important; sometimes more so than efficiency.
  – Facilities to specify time and delays.

• Facilities to interact with special purpose hardware
  – need to program devices in a reliable and abstract way

• Concurrent control of separate components
  – devices operate in parallel
Outline: Handling Complexity

- Examples of Real-Time computation
- Real-time programming languages
- Building large systems through decomposition and abstraction
- Information hiding and separate compilation
- Using Modules
- Summary
Real Time Computation

- Logical versus Real Time

- Embedded computing is often at the junction of two time topologies

- Correctness depends not only on the logical (numerical) results but also on the physical instants at which the results are produced.
Example 1

- **Goal:**
  - Maintain a given flow set point (rate of flow) despite changing environmental conditions.
    - Varying level of the liquid in the vessel.
    - Temperature of the fluid (affecting its viscosity)
- The computer controls the plant by setting the position of the control valve.
- Flow sensor is used to determine the effect of the control.
Real Time Control

• Dynamics:
  – Stability of the control is the main issue.
  – When the control valve must be activated, and by how much influenced by:
    • Valve takes 10 seconds to go from 0% closed (open) to 100% closed (open).
    • Flow sensor’s precision is 1%.
    • Sampling interval 100 msecs.
• Output action by the controller will effect the environment after a delay ($\Delta_1$).
• Observing the effect on the environment will involve a delay introduced by the sensor ($\Delta_2$).
• Measure or derive these delays to implement the temporal control structure.. (discussed later along with delays in PL)
Engine Control

• **Goal:**
  - Calculate the amount of fuel and the moment at which this fuel must be injected into the combustion chamber of each cylinder.

• **Fuel amount and injection time depend on:**
  - Intentions of the driver (position of the accelerator pedal)
  - Current load on the engine
  - Temperature of the engine
  - The position of the piston in the cylinder
  - Many more conditions....
Engine Control

• The dynamics:
  – the position of the piston indicated by the measured angular position of the crankshaft.
  • Precision required: 0.1 degree
  – At 6000 rpm, 10 mscs for each 360 degree rotation.
  – temporal accuracy (sensing when the crankshaft has passed a particular position):
    3 μsecs.
Engine Control

• Fuel injection by opening a solenoid valve:
  – Delay from the time “open” command issued by the computing system and the time at which valve opens:
    • hundreds of $\mu$ seconds!
    • Changes depending on environmental conditions:
      – Temperature, ….
  – This delay is measured each cycle and used to compute when the next “open” command to be issued so that fuel is injected at the right time.

• Extremely precise temporal control is required.
  – Incorrect control can damage the engine!

• Up to 100 concurrently executing software tasks must run in tight synchronization.
Rolling Mill Control

• **Goal:**
  – Roll a slab of steel to a (thin even) strip and coil.

• **Distributed solution.**

• **Dynamics:**
  – The sensor values are read by the sensor computer.
  – The model computer computes new set points (roll dynamics) for the three drives.
  – The control computer sends out the actuating signals to the drives.
Example: Concurrency

- Consider a programmable thermostat consisting of 3 threads

```c
/* monitor temperature */
do forever {
    measure temp;
    if (temp < setting)
        start furnace;
    else if (temp > setting + delta)
        stop furnace;
}

/* monitor time of day */
do forever {
    measure time;
    if (6:00am)
        setting = 72;
    else if (11:00pm)
        setting = 60;
}

/* monitor keypad */
do forever {
    check keypad;
    if (raise temp)
        setting++;
    else if (lower temp)
        setting--;
}
```
Role of Scheduling

Cortadella, et al: Quasi Static Scheduling
Environmental controller

float sample, last;
last = 0;
forever {
    sample = READ(TSENSOR);
    if (|sample - last| > DIF) {
        last = sample;
        WRITE(TDATA, sample);
    }
}

TSENSOR

HSENSOR

TEMP FILTER

HUMIDITY FILTER

TDATA

HDATA

CONTROLLER

AC-on

DRYER-on

ALARM-on
Environmental controller

**TEMP-FILTER**

```c
float sample, last;
last = 0;
forever {
    sample = READ(TSENSOR);
    if (|sample - last| > DIF) {
        last = sample;
        WRITE(TDATA, sample);
    }
}
```

**HUMIDITY-FILTER**

```c
float h, max;
forever {
    h = READ(HSENSOR);
    if (h > MAX) WRITE(HDATA, h);
}
```
Environmental controller

```c
float tdata, hdata;
forever {
    select(TDATA,HDATA) {
        case TDATA:
            tdata = READ(TDATA);
            if (tdata > TFIRE)
                WRITE(ALARM-on,10);
            else if (tdata > TMAX)
                WRITE(AC-on, tdata-TMAX);
        case HDATA:
            hdata = READ(HDATA);
            if (hdata > HMAX)
                WRITE(DRYER-on, 5);
    }
}
```
“Generalized” Embedded Computer: Another View

- Real Time Clock
- Database
- Operator’s console
- Algorithms for digital control
- Data logging
- Data retrieval & display
- Operator interface
- Interface
- Engineering System
- Remote Monitoring System
- Display
RT Programming Languages

• Sequential systems implementation languages
  – Assembly languages
  – RTL/2, Coral 66, Jovial, C.
  – these require support from the operating system for RT modeling

• Concurrent languages
  – Ada (Ada95), Modula, C/C++ extensions, RT POSIX, Occam2

• An important aspect of RT programming languages is that these include facilities to interact directly with the underlying hardware
  – unlike general purpose programming languages that rely on OS abstractions.
OS Interaction

- User program
- Operating system
- Hardware

- Systems programming language or assembly language
- General purpose programming language

- User program including operating system component
- Hardware

- Real-time programming language
Programming Issues by Example

- We shall consider three different languages to highlight important aspects of RT programming
  - Ada
    - Used in safety critical systems
    - A language in which concurrent programs can be written
  - C
    - Because it is C (popular)
    - A sequential language, concurrency through OS processes
  - OCCAM2
    - Language no one uses, (but should?)
    - Modeled after CSP
    - A concurrent programming language

- Caveat: You can not expect to learn any programming language without programming at least a few thousand lines of code in it.
  - Sub-caveat: not much attention to syntax, though it is important in practice.
Style

• All three are **block structured**
  – Description consists of blocks
  – What blocks represent varies

• For instance, a block in ADA consists of
  – Declaration of objects
  – Sequence of statements
  – Collection of exception handlers

• A block in C consists of
  – Declarations
  – Sequence of statements
Ada

• An Ada program consists of one or more program units
  – a subprogram (procedure or function) -- can be generic
  – a package (may be generic) used for encapsulation and modularity
  – a task -- used for concurrency
  – a protected unit - a data-oriented synchronization mechanism

• Library units
  – package, subprogram

• Subunits
  – subprogram, package, task, protected unit.
Blocks

declare
  - definitions of types, objects
  - subprograms etc.

begin
  - sequence of statements

exception
  - exception handlers
end;

*a block can be placed anywhere a statement can be placed.*
Scopes

- Objects declared in a block may only be used in that block
- any statement in the sequence of statement may itself be a block
- exception handlers can be used to “trap” errors

```plaintext
function Largest(X: Vector) return Integer is
    max: Integer := 0;
begin
    for I in X'range loop
        if X(I) > Max then
            Max := X(I);
        end if;
    end loop;
    return Max;
end Largest;
```
C

- Sequential, main structuring through functions (& files)
- Block structured (called compound statements)

{  
  <declarative part>
  <sequence of statements>
}

- declarative part can not contain functions
- sequence of statements can contain compound statements
Occam2

- Process oriented instead of statement oriented
- Sequence of process is equivalent to sequence of statements
- Syntax:
  - each process on a separate line
  - declaration terminates with a colon (:) 
  - indentation is important

SEQ i := 0 FOR 10
A[i] := 0
WHILE BOOL
SEQ
....

INT Max IS 10: -- constant
[Max] Real32 Reading : -- array 0 .. 9

INT tmp:
SEQ
  tmp := A
  A := B
  B := tmp

----------------------------------------
IF
  A <> 0
    IF
      B/A > 10
        high := TRUE
        TRUE
        high := FALSE
    TRUE
    SKIP

Last two lines make sure that IF does not become STOP when a is 0.
Data Types

- All three languages (Ada, C, Occam2) manipulate objects abstracted from their actual hardware implementation.
  - Ada: constants, types, variables, subprograms, packages
  - C: constants, types, variables, functions
  - Occam2: INT, INT16, INT32, INT64, BOOL, BYTE
    - more restrictive, since user-defined types are not allowed.
- Ada, Occam2 are strongly types, C is a “bit loose”
- Enumerated types
  - C: constants must be uniquely defined
  - Ada: names can be overloaded
  - both languages allow manipulation of enumerated types
    - C by the standard integer operations
    - Ada by use of attributes
Dealing with Numbers

• Real life = Real numbers
• Traditional approach in microprocessors
  – Extend the range of the numbers with special purpose hardware, Floating point units
  – FP hardware is typically not available in microcontrollers
    • (And even in some high-end scalable multiple processor fabrics)
• 10 bit ADC sampling a temperature sensor: 0.00-50.00 deg C
  – ADC range: 1-1022 → Display range: 0→5000
  – Higher display precision than ADC (5000/1021=4.897)
    • Will not show all values in the display range
    • Increase ADC bits or lower display precision
• Scaling operations
  – Implement scaling through shift operations
    • ADC*0.4897 for display range: 000→500
    • = ADC *4897 / 10000 + 5000/10000 (for rounding)
  – Or as table lookup.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1022</td>
<td>50.00</td>
</tr>
<tr>
<td>1023</td>
<td>--</td>
</tr>
</tbody>
</table>
Types Continued

• Real Numbers
  – can be represented either as a floating-point number
    • $M \times R^E$
    • divergence between a floating point number and its real value is “relative error” (related to the size of the number)
    • single or double precision (built in)
  – or as a scaled integer that implements a fixed-pint number
    • a product of an integer and a scale
    • scales are known at compile time
    • divergence between a scaled integer and its real value is an “absolute error”
    • user defined arithmetic (i.e., left to the programmer)
• Ada, C use “float” for implementation dependent (sp) fl
• Occam2 “reals” are REAL16, REAL32, REAL64 that explicitly specify the number of bits
Example: Ada

- **FP numbers**
  - `type New_Float is digits 10 range -1.0E18..1.0E18;`
- **A subtype of this type can restrict the range or the precision:**
  - `subtype Crude_Float is New_Float digits 2;`
  - `subtype P_New_Float is New_Float range 0.0..1000.0;`
- **Fixed point numbers as scaled integers**
  - `type Scaled_Int is delta 0.05 range -100.00..100.00;`
    - defines absolute error bound “delta” of 0.05
    - the number of bits can be calculated from the range and delta
      - five bits for the fraction (1/20) (nearest is 1/32)
      - eight bits for the range (including sign)
    - `<sbbbbbbb.fffff>`
  - Specification of “delta” and “range” frees up programmer from implementation details regarding scaling integers
Structured Data Types

- Ada, C: arrays and records
- Occam2: arrays

- INT Max IS 10:
  - a constant

- [Max] REAL32 Reading:
  - Reading is an array with 10 elements.

- Dynamic Data Types:
  - requires a memory allocation facility at runtime, has overheads
  - Occam2 does not provide any dynamic data structures
  - C allows pointers to objects
  - Ada: an “access” type is used (similar to pointer)
Example

type Node;
type Ac is access Node;
type Node is
    record
        Value: Integer;
        Next: Ac;
    end record;
V: Integer; A1: Ac;
begin
    A1 := new Node;
    A1.Value := V;
    A1.Next := null;
    ...

Building ES Software Systems

• Decomposition
  – systematic breakdown of a system into smaller parts
  – supported by modules, packages, separate compilation
  – “top-down” design

• Abstraction
  – allows detailed consideration of components to be postponed yet enables the essential part of the component to be specified
  – “bottom-up” design
How to decompose large systems?

• Decompose them into “Modules”
  – a collection of logically related objects and operations
  – use encapsulation

• Encapsulation
  – technique of isolating a system function within a module with
    a precise specification of the interface

• With modules, one can support
  1. information hiding
  2. separate compilation
  3. abstract data types
1 Information Hiding

• A module structure supports reduced visibility by allowing information to be hidden inside the body
• The specification and body of a module can be given separately
• Ideally, the specification should be compilable without the body being written
  – e.g., in Ada, there is a package specification and a package body; and a formal relationship between the two. (Compile time error detection)
• In C, modules are not so well formalized.
  – Use separate header (.h) files to contain the interface to a module and a .c file for the body. No formal relationship, therefore errors are caught at link time.
  – Modules are not first class language entities.
Module Specification and Body

- **Specification**
  - Only objects declared here are visible externally
  - May also be a placeholder for verification checks
    - Assertions, invariants (a la, passive process in VHDL)
- **Ada uses “Open Scoping”**
  - All identifiers visible at the point of the package declaration can be accessed within the package
  - For access outside the package, use prefixing by the package identifier (“MyPackage.Function”)
    - Can simplify it by using “use” declarations.

- **Even though important, Modules are not first class entities**
  - Module types cannot be defined; pointers to modules cannot be created, and so on.
Example

Package Queuemod is
    function Empty return Boolean;
    procedure Insert (E:Element);
    procedure Remove (E:out Element);
End Queuemod;

Packagebody Queuemod is
    type Queue_Node_T;
    type Queue_Node_Ptr_T is access Queue_Node_T;
    type Queue_Node_T is
        record
            contents: Element;
            Next: Queue_node_Ptr_T;
        end record;
    end record;

...
2 Separate Compilation

- Modules can be compiled separately (into a library)
- Both in Ada and C, a package (module) specification and body can be (pre)compiled
- A library can access another library
  - Using with “with” clause or “#include”
- Separate compilation supports bottom up programming
  - Useful in a top down design as well (via specification & body)
  - Ada supports it further by “is separate” clause
    - (that allows a procedure to be added later until the main program has been defined)
- In Ada, separate compilation is supported by the language
  - That is, its typing rules apply equally across libraries
- In C, comprehensive type checking is not possible
  - Though one can lint it.
Aside: Reflection in Java

- Data, meta-data and reification
- Ultimately, ability by a class to examine itself
  - Ability of a program to obtain information about the functions defined within the program
- Reflection is used by programs that need to examine or modify the runtime behavior of the applications (running on JVM)
  - Requires runtimes resolution of types, runtime permissions
- Supported through reflection API
  - An application may make use of external, user-defined classes by creating instances of extensible objects using their names
  - Methods for browsing classes, debugging and test.
import java.lang.reflect.*;

public class DumpMethods {
    public static void main(String args[])
    {
        try {
            Class c = Class.forName(args[0]);
            Method m[] = c.getDeclaredMethods();
            for (int i = 0; i < m.length; i++)
                System.out.println(m[i].toString());
        }
        catch (Throwable e) {
            System.err.println(e);
        }
    }
}
3 Abstract Data Types

- Extension of the idea of separation of specification from (physical) implementation

- A module can define both a type and the operations on the type

- Details of the type must be hidden from the user
  - Structure of the ADT is within the module specification

- Since more than one instance of the type can be supported
  - Need a ‘create’ routine in the interface of the module

- As modules are not first class, the type must be declared
  - and instances of the type passed as a parameter to the operation.
ADT and Separate Compilation

• ADT is complicated by the requirement for separate compilation of a module specification from its body
  – Since the structure of the ADT is meant to be hidden, it is logically in the module body
  – But then the compiler would not know the size of the type when using only its specification?

• Remedy: force the programmer to use a level of indirection
  – To ensure the user is not aware of the details of the type, it is passed as a pointer (as in C).
  – An incomplete declaration of the type is given in the .h file.
Example: A Queue module interface in C

typedef int element

struct queue_t
typedef struct queue_t *queue_ptr_t;

queue_ptr_t create();

int empty(queue_ptr_t Q);

void insertE(queue_ptr_t Q, element E);

void removeE(queue_ptr_t Q, element *E);

#include “queue.h”

struct queue_node_t {
    element contents;
    struct queue_node_t *next;
};

struct queue_t {
    struct queue_node_t *front;
    struct queue_node_t *back;
};

queue_ptr_t create()
{ ...
}
ADT and Separate Compilation

• While forcing a pointer to the structure of an ADT type in the module body works, it is not always clear

• Ada also allows part of the implementation to appear in the specification
  – But it is accessible only from the package body
  – It is called the “private” part of the specification.
Package Queuemod is
    type Queue is limited private;
    procedure Create (Q: in out Queue);
    function Empty (Q: Queue) return Boolean;
    procedure Insert (…)
    procedure Remove (…)
Private
    -- none of the following declarations are externally visible
    type Queueptr is access Queuenode;
    type Queuenode is
        record
            contents: processID;
            next: Queueptr;
        end record;
    type Queue is
        record
            Front: Queueptr;
            Back: Queueptr;
        end record;
End Queuemod;

-- limited private means that only those subprograms defined in this package can be applied to the type;
-- else use “private” that enables overloading elsewhere.

Package body Queuemod is
    ---
End Queuemod;
OOP and abstract data types

• OOP includes
  – type extensibility (inheritance)
  – automatic object initialization (constructors)
  – automatic object finalization (destructors)
  – run-time dispatching of operations (polymorphism)

• Ada95 supports the above through tagged types and class-wide programming
• C++ supports OOP through the use of classes.
Summary

• Module supports
  – information hiding, separate compilation, and ADTs
• Ada and C have a static module structure
• C informally supports modules; C++ has a dynamic module structure called a class
• Both packages in Ada and classes in C++ have well-defined specifications which act as the interface between the module and the rest of the program.
• Separate compilation enables libraries of precompiled components to be constructed.

Next lecture: fault tolerance, exceptions, concurrency.