C++ for Embedded Systems

ACCU 2008
Detlef Vollmann

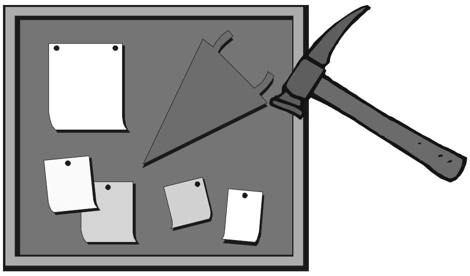
C++ for Embedded Systems

Detlef Vollmann
Siemens Building Technologies
Zug, Switzerland

eMail: detlef.vollmann@siemens.com
dv@vollmann.ch

A Warning

- You will (hopefully) learn some useful techniques in this tutorial.
- Use them, if you need them.
- Don't use them, if you don't need them.
  - A hammer is a useful tool if you want to put a nail into a wall.
  - It's not so useful, if you want to place a pin on a corkboard.



C++ for Embedded Systems Copyright © 1995-2008 Detlef Vollmann 3

Overview

Introduction

Base concepts

C++

Language costs, benefits; SL/STL

Examples

Typical systems

Architectures, Patterns, Idioms

Useful techniques

C++ for Embedded Systems Copyright © 1995-2008 Detlef Vollmann 4

Introduction

Object Orientation

What is it about?

C++ History

Why C++ is designed for embedded systems.

Embedded Systems

What are embedded systems?

Interface Bilding

- Well defined interfaces provide better maintainability
  - easier variants
  - easier replacements of modules
  - easier adaptions to new situations
- Inheritance means interface bilding
  - interface inheritance vs. implementation inheritance

The Foundation of OO

- Interfaces
  - make your responsibilities clear
- Encapsulation
  - protected implementation
  - hidden implementation
- Well…

```
#define private public
```

  - this is C++
- This is all about modularization.

The Golden OO Rule

"One class per abstraction"

- This produces a quite fine-grain modularization.

- Secondary rule:

"Keep your classes independent."

  - Use composition and delegation instead of inheritance.
- These rules produce really flexible systems.
- If properly applied, they introduce no performance overhead.

C++ History

- C++ was designed from the beginning as a system programming language.
- C++ was designed to solve a problem – a complex, low (system) level one.
- Design goals:
  - Tool to avoid programming mistakes as much as possible at compile time
  - Tool to support design – not only implementation
  - C performance
  - High portability
  - Low level
  - Zero-overhead rule ("Don't pay for what you don't use.")

C++ Language Costs

- "TASATAFL"
- Generally, C++ is as fast as hand-coded assembler
  - but no rule without exception
- Abstraction mechanisms sometimes cost
  - program space
  - runtime data space
  - runtime performance
  - compile-time performance
- Non-abstraction solutions cost as well

Java / C#

- Generally same costs as C++
- Plus some additional costs
  - more dynamic memory management
  - GC (sometimes faster)
  - virtual machine
- No real hardware access
  - interrupts
  - registers
- No generic programming

C

- Sometimes there is no C++ compiler available :-(
- Still use OO design
  - at least partly
  - better modularity
- Implementation much harder
- Typically same costs as C++

Embedded Systems

- Device
  - not hardware and software
  - No computer aware operator
- You control everything
  - hardware
  - operating system
  - all application level software

Embedded Systems

- Reliability
  - Web service providers are very proud about 99.9% uptime – that means about 8.5 hours downtime per year.
  - Sorry, but that's not good enough for a lot of embedded systems.
- Robustness
  - Embedded systems don't have a computer-aware operator – if something fails, they must go back to a stable state themselves.
- Constraints
  - CPU cycles (time efficiency)
  - Memory (space efficiency)
  - Power consumption

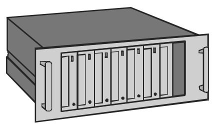
Embedded Systems

- Growing complexity
  - User interactions
  - Runtime-configurability
- Growing demands
  - Functionality
- Growing resources
  - We can afford better tools.

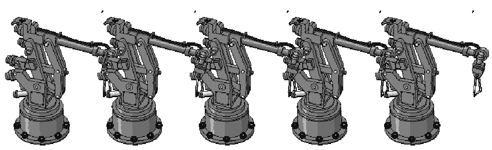
Embedded Systems

- Small systems


  - high volume, low production costs, larger development costs
- Medium systems



- Large systems



  - custom built, flexibility, time to market

Embedded Systems

- Real-Time
  - If we miss a deadline, something goes wrong.
  - No matter whether μs, ms, s, min, ...
- Special hardware
  - In most cases, we have to write our own device drivers.
- Special memory types
  - ROM, EEPROM, Flash (NOR/NAND), NVRAM, SRAM, DRAM, VRAM...
- Concurrency
  - We do everything, and we have to coordinate it.

Concurrency

- Multi-Processing
  - often you have more than one processor in your system
  - specialized co-processors
- Know your hardware
  - caching
  - instruction re-ordering
  - interrupt handling
  - hardware locks, dual-ported RAM, ...
- Know you OS
  - mechanisms
  - driver frameworks

Overview

Introduction
- Base concepts

C++
- Language costs, benefits; SL/STL

Examples
- Typical systems

Architectures, Patterns, Idioms
- Useful techniques

C++

Language costs
- What's the price we have to pay for inheritance, exceptions, ...?

Language benefits
- What do we actually get that's worth the price?

S(T)L
- Can we use what's already there?

Language Costs

- "TASATAAFL"
- "No" costs
  - This is really cheap :-)
- "Hidden" costs
  - The small print: you can see it, but it's quite hard to find.
- Really hidden costs ("compiler magic")
  - This is the real price.

"No" Costs

- Well, even longer identifiers cost...
- There are a lot of myths about all the new stuff in C++ out there but here are the facts:
  - Namespaces
    - are just like longer names.
  - New-style casts
    - `static_cast<T>`, `const_cast<T>`, `reinterpret_cast<T>` cost just the same as their C counterpart `(T)`.

"Hidden" Costs

`A a; B b; C c = a; b = a + c;`

Constructors / destructors

Conversions

Overloaded operators

Default arguments

Inline

Inheritance

Templates

Constructors / Destructors

```
class A
{
    size_t size;
    unsigned char *data;
    static const size_t default = 100;
public:
    A() : size(default), data(new unsigned char(default)
    { for (size_t i = 0; i != size; ++i) data[i] = 0; }
    ~A();
    A& operator=(A const &);
};
A a1, a2;  a1 = a2;
```

- Rule: "Make default constructors as cheap as possible - but not cheaper."

Conversions

- Even in C:
  - `int i = 5; double d = i;`
- In C++ much more important:
  - `f(string const &); f("abc");`

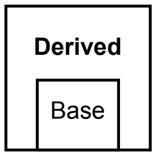
Overloaded Operators

- Even in C:
  - ```
    struct S { int buffer[2048]; };
    struct S s1, s2;
    s1 = s2;
    ```
- In C++, even harmless looking operators can be expensive:
  - `a->b`
  - `a < b`
  - `i++`
- `for` loops in C and C++ look different – for a reason:
  - `for (i=0; i<max; i++)`
  - `for (i=0; i!=max; ++i)`
    ```

Default Arguments

- If parameter passing can be expensive, default arguments can be expensive – and they are hidden
- ```
  f(int i,
    double d,
    string const & s = "abc");
  ```


Inline

- Inline functions can greatly improve time efficiency.
- Inline functions often degrade space efficiency.
- Inline functions can even degrade time efficiency!
  ```

Inheritance

```
class Base { unsigned int bigBuffer[bigSize]; };
class Derived : public Base { ... };
```

- Inheritance is containment

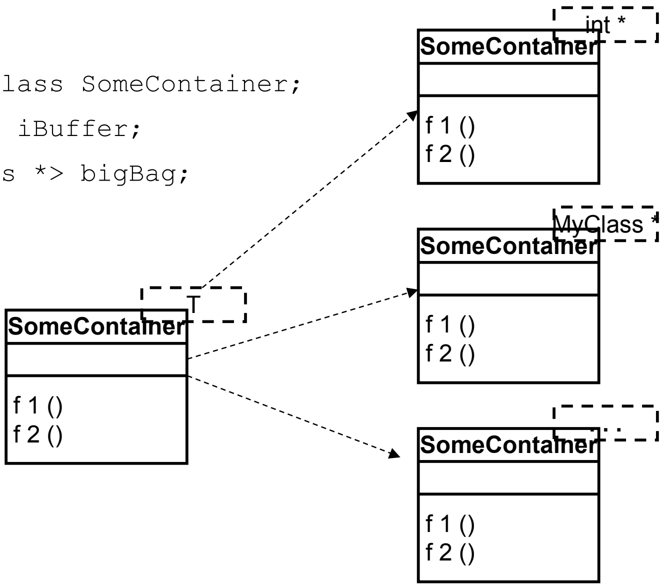


- If there is no absolute 1:1 relationship, use delegation and reference for common data intensive components.

Templates

- "Code bloat"

```
template <class T> class SomeContainer;
SomeContainer<int *> iBuffer;
SomeContainer<MyClass *> bigBag;
...
```



Templates' "Code Bloat"

- Smart compilers avoid this by comparing the code.
- Smart library writers avoid this by explicit specialization:

```
template <class T*> class SomeContainer
{ implementation using void* };
```

- Programmers can avoid it by thinking about their specializations:
- Not:

```
Buffer<int, 255> b1;
Buffer<int, 256> b2;
```

- but:

```
Buffer<int, 256> b1, b2;
```

C++ for Embedded Systems 31

"Hidden" Costs – Summary

- C++ hides a good amount of details.
- This is good – the code is easier understood.
- But be aware of the hidden work!

"Know your language!"

"Know your libraries!"

C++ for Embedded Systems 32

Really Hidden Costs

- "Compiler Magic"

Virtual Functions

Multiple Inheritance

Exceptions

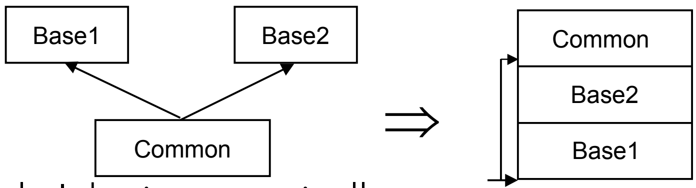
Dynamic casts

Virtual Functions

- Virtual functions typically cost:
  - one `vtable` per class
  - one `vptr` per object (per inheritance branch)
  - one table lookup (a pointer dereference and a small addition) per virtual function call

- In most cases, a virtual function call can't be inlined.

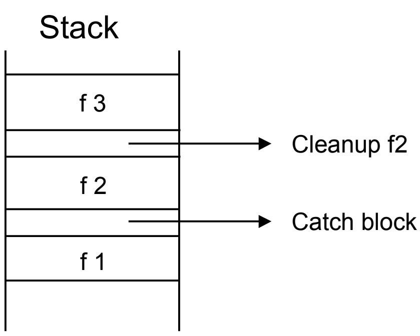
Multiple Inheritance



- Multiple Inheritance typically costs:
  - one `this`-pointer adjustment for each conversion to a non-first base class
  - an additional `vptr` for each additional base class branch
  - an additional destructor entry for each additional base class branch
- Virtual Inheritance
  - worse, but not really bad
  - avoid non-static data members in virtual base classes

Exceptions

- Exceptions cost.
- They even cost when no exceptions are thrown, as long as your compiler doesn't know that for sure.
- Exceptions don't cost a lot.



Exceptions

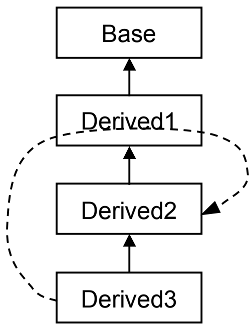
- There are alternatives for exceptions
  - e.g. www.vollmann.ch/en/pubs/cpp-excpt-alt.html
- Exceptions alternatives cost as well.
- If you don't throw exceptions, tell your compiler:
  - project wide: compiler flag
  - locally: `throw()` - specification
    - but in most cases this doesn't really help performance-wise
  - both: `NOTHROW`

```
#if EXCEPTIONS_ENABLED
  #define NOTHROW : throw()
#else
  #define NOTHROW
#endif
```

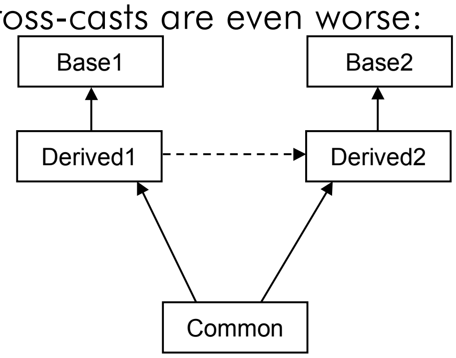
- BTW: Make your code exception safe anyway!

Dynamic Casts

- Simple dynamic down-casts cost more than often expected.



- Cross-casts are even worse:



Summary Real Hidden Costs

- Again: "Know your language."
  - and know your implementation of C++
- You should be aware of the costs of your design decisions – and spend them, if the benefits are worth it.
- "Don't over-optimize."
- In most cases, the non-C++ alternatives have higher overall costs.

Benefits

C++ lets you use OO directly, with all its benefits:

Reliability

It runs, and runs, and runs ...

Reusability

Special versions, different hardware and similar systems

Reliability

Smaller Units

Small is beautiful.

Cleaner Code

Ease the code review.

More Robust Code

Let the compiler do the work!

Smaller Units

- Classes are protected units.
  - Nobody can change (or access) your data without your control.
  - Users of your class are constrained to the published interface.
- Classes have explicit interfaces.
  - You can change the implementation.
  - You can substitute a class by your own version.
- Classes are self-contained.
  - You can re-use them elsewhere.
  - Again: you can substitute them.
- Classes are plugged into frameworks.
  - Re-use complete architectures.

Cleaner Code

- Small units
  - In smaller, self-contained units, mistakes are much easier to spot.
- Clear responsibilities
  - From the published interface, it's clear what you have to do – and what's an SEP.
- Clear delegation
  - If something is not your problem, it's clear who else is responsible for that.

More Robust Code

- Automatic initialization
  - Nobody can forget to make a clean start – the compiler cares for you.
- Automatic cleanup
  - Never again forget to free your locks or your memory – again the compiler (together with useful library classes) cares for you.
- Protected separations
  - The compiler enforces your boundaries.

Reusability

- Classes are easier to re-use than functions (not easy!)
  - Self containment (enforce this!)
  - Clear responsibilities
- Plug-in components into framework.

Reusability

- Reusability for embedded systems is often much easier (and more important) than for desktop systems
  - Special versions
    - A customer wants some of the functionality a littlebit different.
  - Different hardware
    - For embedded systems, porting is often the daily work:
      - different components to drive
      - new hardware line
      - new microcontrollers
  - Similar systems
    - If you write the software for one microwave, chances are good that you have to write one for a different model.

Summary Benefits

- Though the OO (and C++) mechanisms sometimes cost you a bit, the benefits nearly always outweigh the costs:
  - You create your systems faster (through less debugging and more re-use).
  - You create more reliable systems (due to cleaner code).
  - Your systems are more flexible and therefore the time to market for variations is much shorter.

Standard (Template) Library

- In general, the performance (time and space) of the C++ standard library is really good.
  - You can hardly beat it with your own implementation!
- But for embedded systems, there are some issues to look at:

Dynamic memory allocation

Embedded systems are different.

Strings

I/O streams

Internationalization (i18n)

STL

Dynamic Memory Allocation

- Unknown performance
  - in general quite bad in space and time
- Often wrong functionality:
  - allocation in specific memory region
  - different deallocation mechanism
  - different pools for different objects
  - re-using unfreed objects

Dynamic Memory Allocation

- Should you really use it?
  - In low-level classes: No!
  - At application level: it depends.
  - Most space can be pre-allocated (at compile-time or system initialization).
  - There is nothing inherently wrong with dynamic memory allocation – but it can fail.
    - Either make sure it can't fail.
    - Or care for failure.

Strings

- Heavy-weight objects
  - dynamic memory allocation
  - high functionality
  - problematic reference counting
- For fix-length string literals, `char *` (or `wchar_t *`) is just fine.
- For dynamic length, non-changing strings (obtained at runtime) you might write your own class (with sharing semantics).
- For changing strings, try to live with `std::basic_string<>`.

I/O streams

- Very useful abstraction
- Unfortunately, most implementations have quite bad performance (space and time).
  - Main problem is internationalization formatting overhead.
- For low-level classes, don't use standard I/O streams.
- For application level classes, standard C++ I/O streams can be very useful.
  - But look for an efficient implementation (and test it before you really use it).

Internationalization

- Very useful for a lot of embedded systems.
- Again, most implementations have quite bad performance (space and time).
- Use good implementation or implement your own subset.

Standard Template Library

- Containers
  - store your objects
- Iterators
  - access your objects
  - not useful for concurrency
  - use internal iterators
    - `bag.find()` instead of `find(bag.begin(), bag.end());`
- Algorithms
  - work with your objects
  - based on iterators
  - useful for implementing internal iterators

Containers

- Always dynamic memory allocation
  - for fixed-length containers, use just plain arrays.
- Not so easy-to-use allocator interface.
  - Learn it.
- No useful interface for concurrency

```
while (bag.empty()) sleep(1);
doSomething(*bag.begin());
```

- For objects used concurrently, write your own containers.

```
T *object = bag.getItemExclusively();
doSomething(*object);
bag.returnItem(object);
```


Summary Library

- Very useful components.
- Most of them not suitable for low-level use.
- For application level usage, don't write your own when the standard components fit the bill.
- Possibly you have to provide your own allocators.
- STL components are not suitable for concurrency (by interface definition).
- Test I/O stream and I18n performance in advance.

Embedded Design Rules

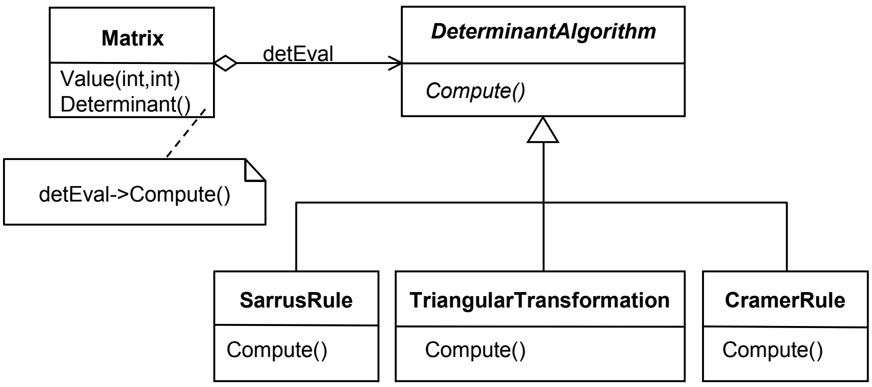
- Some design rules are particularly important for embedded systems:
- Don't tap into the OOAD trap: Analyzing a system top-down in classes and implement them.
- Use a bottom-up lego approach.
- Most XP concepts are usefully applicable to embedded systems design.

Interfaces in C++

- Virtual functions
  - classic OO technique
  - runtime flexibility
- Templates
  - templates allow for implementation "inheritance".
  - templates allow for interface inheritance.
  - templates allow for high flexibility combined with high efficiency.
  - templates are compile-time constructs.
  - beware of "code bloat"!

Interfaces

- The Strategy pattern shows how alternative algorithms can be implemented:



Virtual Functions

```
class Matrix
{
  public:
    //...
    float &value(int, int);
    float determinant() { return detEval->compute(this); }
  private:
    DeterminantAlgorithm *detEval;
    // ...
};

class DeterminantAlgorithm
{
  public:
    // ...
    virtual float compute(Matrix*) = 0;
};
```

Templates

```
template <class DeterminantAlgorithm>
class Matrix
{ public:
    //...
    float &value(int, int);
    float determinant() { return detEval.compute(this); }
  private:
    DeterminantAlgorithm detEval;
       // ...
};

class SarrusRule
{ public:
    // ...
    float compute(Matrix*);
};
```


Summary Design with C++

- C++ allows different design styles.
  - it's a multi-paradigm language
- Decisions must be based on concrete needs.
- Decisions should be consistent throughout the project.
- But they don't need to be the same.

Overview

Introduction

Base concepts

C++

Language costs, benefits; SL/STL

Examples

Typical systems

Architectures, Patterns, Idioms

Useful techniques

Embedded Systems Technically

- Small systems
  - 8- or 16-bit, microcontrollers, SOC
  - No OS, own OS or small RTOS
- Not quite so small
  - 16- or 32-bit, microcontrollers, external memory
  - RTOS with single memory address space

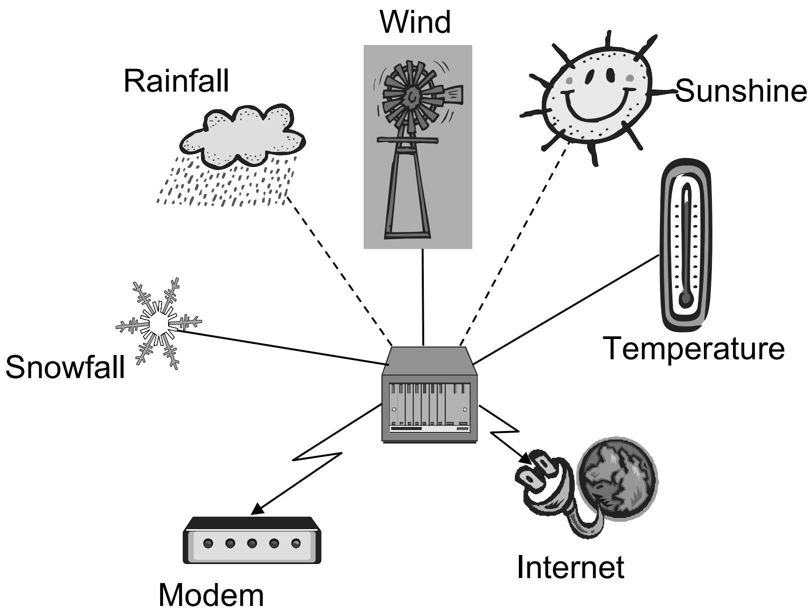
Embedded Systems Technically

- Medium systems
  - 32- or 64-bit, MMU, microcontrollers, external memory
  - (RT)OS with protected processes, virtual memory, POSIX-conformant
- Large systems
  - No HW constraints
  - Standard OS with real-time extensions

RT Tester

- A simple device to test real-time latencies
  - sends signals and measures the response time
- 8-bit microcontroller
  - 8KB flash, 512 Bytes EEPROM, 512 Bytes RAM, 16MHz
- Simple scheduler
  - co-operative with ISRs
- 4 tasks
  - control communication
  - LED control
  - signal latency test
  - SPI latency test

Weather Station



Weather Station

- Typical data-concentrator
- Instrument I/O: serial and pulse
- Communication I/O: serial (modem) and Ethernet (wireless, satellite), Internet
- DRAM, Flash, CardBus
- Embedded Linux

Overview

Introduction

- Base concepts

C++

- Language costs, benefits; SL/STL

Examples

- Typical systems

Architectures, Patterns, Idioms

- Useful techniques

Architectures

Frameworks

Layers

Pipes and Filters

Micro-Kernel

Scheduler

Frameworks

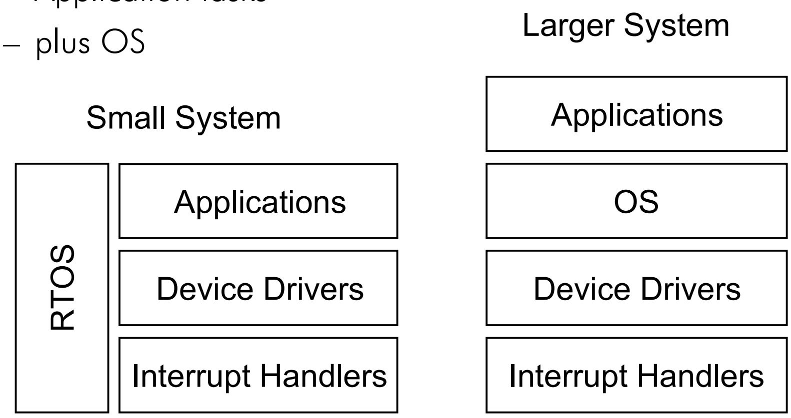
- Frameworks provide you a full environment where you can plug-in your specific objects.
- General control flow is provided by framework.
  - Callback style of programming
  - Event-driven programming
- System architecture is completely provided by framework.
- Design of your application classes is mainly pre-defined.
- Frameworks allow for adaptation to your specific application at variability points.

Framework Numbers

- Sample framework:
  Satellite Attitude and Orbit Control System (AOCS)
- Typical application size: < 500KB code
- Framework overhead:
  - Time: < 1%
  - Space: < 15%
- Framework still hard real-time.

Layers

- 3+1
  - Interrupt handlers
  - Device drivers
  - Application tasks
  - plus OS



Pipes And Filters

- A data concentrator is a typical one-way application:
  - External instruments provide the data
  - One application task per external instrument reads the data (using device drivers)
  - Another task processes the raw instrument data according to instrument configuration.
  - One task combines and correlates the data from the different instruments.
  - A final task transfers the data to a calling central station.

Scheduler

- Small systems without OS
- Still use separate tasks (“One task per functionality”).
- Use co-operative multitasking.
- Don’t write a pre-emptive scheduler yourself.
- Scheduler patterns
  - realtime, non-realtime
  - super loop, multitasking

Patterns

Monitor

  Control your tasks

Containers

  Store your objects

Scheduler

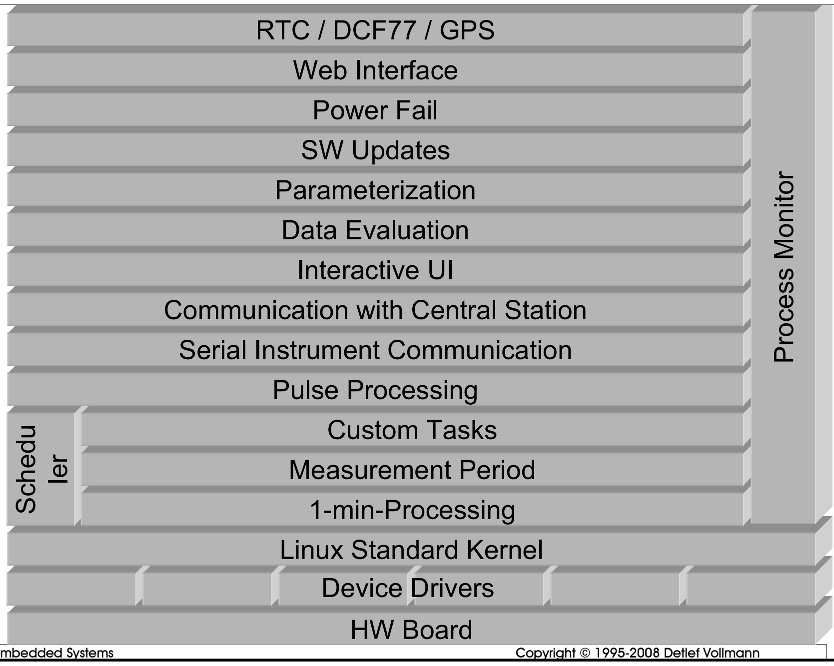
  Wake up your tasks

  Run your tasks

Monitor

- Goal: A robust system
- A Monitor
  - initializes global resources
  - starts your tasks
  - cares for failure
    - Restart of a crashed task on a memory protected system.
    - Reboot on a crashed task on an unprotected system.
- Monitor resembles the init process of Unix systems

Monitor

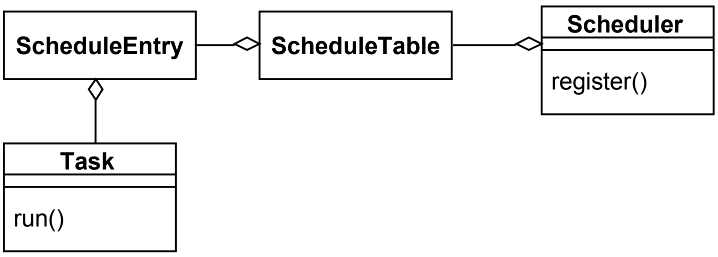


Containers

- Goal: provide safe storage for shared data.
- Provide a concurrency-safe interface according to your needs.
- Make your functions transactional safe, even for crashes.
  - Don't try too much.
  - At least leave a dirty marker.
- Allocator aware
  - Don't do your own memory allocation, but use a provided allocator.
  - Document the allocator usage.
- Example: message queues, property maps, ring buffer

Scheduler

- This is the typical cron process for regular tasks.
- Typical granularity is 1 minute.
- Tasks register themselves.
- Scheduler starts them only dependent on time.
  - No inter-task dependencies.



Summary Patterns

- Components for re-use
- Modules to partition your system design
  - Controlling your tasks
  - Managing your objects
  - Caring for regular jobs
- Only some patterns from two example systems.
- Embedded systems are often quite small, so you need not many patterns for one system.
- But embedded systems are quite diverse, so other systems need other patterns.

Idioms

Memory

In embedded systems, we have to care for the memory ourselves.

IPC / Synchronization

Embedded systems always mean concurrency

Logging

Maintenance support

I/O

This is always special.

Memory

•

Memory access

Memory types

Flash

Memory layers

Checksums

Compression

Encryption

Allocators

Memory saving

Memory Access

- Addressing modes
  - Word size, access method, processor bus, bus width, …
  - Provide a common interface.
- Regions
  - NVRAM, flash, shared memory
  - Provide appropriate allocators.
- Pointers
  - '->' is the common interface to access memory.
- Virtual Memory
- Caching (write back)

Memory Types

- Flash
  - Writing to Flash is special:
    - Sectors
    - Rolling regions due to write-cycle limitations
    - Slow
    - NAND/ECC NOR: only full block writes
- EEPROM
  - Again, writing is special.

Memory Layers

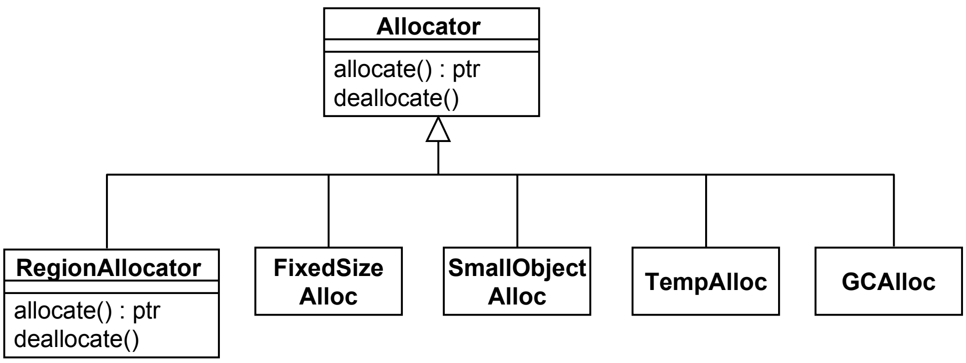
- Doing additional work
  - Compression, encryption, checksums
- Transactional Safe
  - On a crash, your persistent memory must be consistent.

Allocators

- Regions
  - Sometimes it is quite important where your dynamic objects are.
- Strategies
  - Nobody is perfect.

Allocators

- Minimum interface for your own containers
  - STL compatible



STL Allocators

- For use with the C++ standard library, a more complex interface is required.
- Beware of STL implementations
  - make sure that your allocator is called according to your restrictions, and be aware that your solution might not be portable across library implementations.

Memory Saving

- There is a lot of literature on this topic.
- Specifically: Noble/Weir “Small Memory Software”

InterProcess Communication (IPC)

- IPC primitives are much easier and less error-prone to use through an object interface.
- Provide your own class wrappers around the IPC primitives, if your OS provides those primitives.
  - Otherwise implement them.
- Tasks
- Synchronization / Locking
- Shared Memory
- Signals / Events
- Message Queues

Logging

- For whom?
  - operator, service personnel, developer
- What?
  - fatals, errors, warnings, debugging infos
- How much detail?
  - again: for whom?
- Where?
  - space constraints, flash cycle constraints
  - NOR vs. NAND
    - know your hardware
- Time synchronisation

I/O

- Provide a common read/write-interface.
- POSIX provides one (`open, close, read, write, ioctl`).
- Otherwise, you have to provide it.

- This is essentially the UNIX device driver concept, and it's worth to copy it.

- You might provide a `streambuf` for usage with `iostreams`.

Summary Idioms

- Little helper classes help you to abstract from things that might change:
  - Memory access
  - Memory usage
  - Multi-tasking / concurrency
  - I/O

- These are typical components for re-use.

Overview

Introduction

Base concepts

C++

Language costs, benefits; SL/STL

Examples

Typical systems

Architectures, Patterns, Idioms

Useful techniques

Embedded Design

- Constraints
  - Memory, performance, real-time
- Well known environment
  - You can plan in advance
- System programming
  - Low-level
  - Resource management
  - Multi-tasking
    - possibly multi-processing

Embedded Objects

- Object-Oriented Programming often uses a lot of objects
  - short-lived
  - heap-based (at least partly)
  - dynamic memory allocation
- Dynamic memory allocation is often a problem in embedded systems
  - non-deterministic runtime
  - may fail

Embedded Objects

- In embedded systems, OO must be used carefully
  - mechanisms depending on architectural level
  - special "libraries" for specific needs
  - always think about consequences
- Golden optimization rule ("Don't optimize now") only partially true
- Don't use OO for OO's sake
- Use dynamic memory allocation carefully

C++ Wrap-Up

- Use C++ as a better C compiler (zero-overhead rule).
- You don't pay for using C++, but for using the features of C++!
- Real C++ features (OO and genericity) cost something – and offer a lot.
- Know about the costs – and avoid them **where necessary.**
- Use C++ - and OO - techniques to build better systems :-)
- Don't re-invent the wheel.

Questions

¿

References

- Andrei Alexandrescu: “Modern C++ Design”, Addison-Wesley 2001, ISBN 0-201-70431-5
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: “Pattern-Oriented Software Architecture - A System Of Patterns”, Wiley 1996, ISBN 0-471-95869-7
- Andrew Koenig, Barbara E. Moo: “Accelerated C++”, Addison-Wesley 2000, ISBN 0-201-70353-X
- Donald E. Knuth: “The Art of Computer Programming”, Vol. 1 “Fundamental Algorithms” (2nd ed.), Addison-Wesley, 1973

References

- Stanley B. Lippman: “Inside the C++ Object Model”, Addison-Wesley 1996, ISBN 0-201-83454-5
- Scott Meyers: “More Effective C++”, Addison-Wesley 1996, ISBN 0-201-63371-X
- James Noble, Charles Weir: “Small Memory Software”, Addison Wesley 2001, ISBN 0-201-59607-5
- Michael J. Pont: ”Patterns for Time-Triggered Embedded Systems”, ACM Press 2001, ISBN 0-201-33138-1
- Bjarne Stroustrup: “The Design and Evolution of C++”, Addison-Wesley 1994, ISBN 0-201-54330-3
- Herb Sutter: “Exceptional C++”, Addison-Wesley 2000, ISBN 0-201-61562-2

References

- ISO: “Technical Report on C++ Performance”, http://standards.iso.org/ittf/PubliclyAvailableStandards/c043351_ISO_IEC_TR_18015_2006(E).zip
- AOCS homepage: http://control.ee.ethz.ch/~ceg/AocsFramework/
- Boost C++ Libraries http://www.boost.org/
- Dietmar Kühl's C++ Run-Time Library: http://www.dietmar-kuehl.de/cxxrt/

defs.hh

```
// common definitions
// $Id: defs.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef DEFS_HH_SEEN
#define DEFS_HH_SEEN

enum { maxTasks = 5 };
template<typename T1, typename T2>
inline T1 min(T1 const &a, T2 const &b)
{
    return (a < b) ? a : b;
}
#endif /* DEFS_HH_SEEN */
```

array.hh

```
#ifndef ARRAY_HH_SEEN_
#define ARRAY_HH_SEEN_

#include <inttypes.h>

template <int elemSize>
unsigned char *elem(uint8_t *base, uint8_t i)
{
    return (base + i*elemSize);
}

template <uint8_t elemSize, uint8_t size>
class ArrayStore
{
public:
    uint8_t *operator[](uint8_t i)
    {
        return elem<elemSize>(d, i);
    }

private:
    uint8_t d[size*elemSize];
};

template <typename T, uint8_t maxSize>
class Array
{
public:
    typedef T* Iterator;

    Array() : sz(0) {}
    ~Array() { chop(0); }
    T &operator[](uint8_t idx) { return *reinterpret_cast<T *>(data[idx]); }
    Iterator begin() { return reinterpret_cast<T *>(data[0]); }
    Iterator end() { return reinterpret_cast<T *>(data[sz]); }

    uint8_t size() const { return sz; }
    void append(T const &t)
    {
        //new (reinterpret_cast<T *>(data[sz++])) T(t); // doesn't work on AVR
        operator[](sz++) = t;
    }

    void chop(uint8_t idx)
    {
        for (uint8_t i = idx; i < sz; ++i)
        {
            // as we can't call constructors, we can't support destructors
            //reinterpret_cast<T *>(data[i])->~T();
        }
        sz = idx;
    }
private:
    ArrayStore<sizeof(T), maxSize> data;
    uint8_t sz;
};
#endif /* ARRAY_HH_SEEN_ */
```

ring.hh

```
// simple ring buffer
// $Id: ring.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $
#ifndef RING_HH_SEEN_
#define RING_HH_SEEN_
template <typename T, unsigned int size>
class RingBuffer
{
public:
    RingBuffer()
      : first(data),
        last(data),
        end(data + size)
    {
    }

    bool empty() const
    {
        return last == first;
    }

    uint8_t free() const
    {
        int f;
        if (first <= last) { f = size - (last - first); }
        else { f = first - last; }
        return f;
    }

    void append(T const *d, unsigned int len)
    { // silently drops excess data
        for (unsigned int i = 0;
             i != len && next(last) != first;
             ++i, last = next(last))
        {
            *last = d[i];
        }
    }

    T const *getNext() const { return first; }
    void popNext() { first = next(first); }
    void clear()
    {
        first = last = data;
    }

private:
    T *next(T *p) const
    {
        ++p;
        if (p == end) p = const_cast<T *>(data);
        return p;
    }

    T *first, *last, *end;
    T data[size];
};
#endif // RING_HH_SEEN_
```

cpu.hh

```
// cpu.hh
#ifndef CPU_HH_SEEN
#define CPU_HH_SEEN

#include <inttypes.h>

class CPU
{
public:
    CPU();
    void goToSleep();
    void reset() { overflow = 0; }
    uint8_t overflowed() const { return overflow; }

private:
    uint8_t overflow;
};
extern CPU cpu;
#endif /* CPU_HH_SEEN */
```

cpu.cc

```
// cpu.cc
#include "cpu.hh"
#include <avr/io.h>
#include <avr/interrupt.h>

namespace
{
enum { OsKHz = ((F_CPU / 1000) / 256), OsHzTolerance = 3, loopMs = 1 };
enum { prescaleCk1=0x1, prescaleCk8=0x2,
       prescaleCk64=0x3, prescaleCk256=0x4,
       prescaleCk1024=0x05 };
}

CPU::CPU()
    : overflow(0)
{
    TCCR0 = prescaleCk256;
    TCNT0 = 0;
    sei();
}

void CPU::goToSleep()
{
    if (TCNT0 > OsKHz*loopMs + OsHzTolerance)
    {
        overflow = TCNT0;
    }
    else
    {
        overflow = 0;
    }
    while (TCNT0 < OsKHz*loopMs) ;
    TCNT0 = 0;
}
CPU cpu;
```

spschedule.hh

```
// spschedule.hh
#ifndef SPSCHEDULE_HH_SEEN
#define SPSCHEDULE_HH_SEEN

#include <stdint.h>
#include "defs.hh"
#include "array.hh"
#include "cpu.hh"

#include "uart.hh"

class Task
{
public:
    //virtual ~Task() {} // avr-g++ doesn't like this
    virtual void run() {}
protected:
    Task() {}
};

class SchedEntry
{
public:
    SchedEntry(Task *t, uint8_t id, uint16_t period, bool reschedule);
    ~SchedEntry() {}

    bool ready() const { return state == readyToRun; }
    void run() { tsk->run(); }
    void reset()
    {
        state = counting;
        delay = periodTime;
    }
    bool periodic() const { return again; }
    void updateDelay()
    {
        if (delay == 0)
        {
            state = readyToRun;
        }
        else
        {
            --delay;
        }
    }
    void setRemove() { remove = 1; }
    bool getRemove() const { return remove; }
    uint8_t getId() const { return id; }

private:
    enum SchedState { readyToRun=0, counting=1 };

    Task *tsk;
    uint16_t periodTime, delay;
    uint8_t id;
    uint8_t again:1;
    uint8_t state:1;
```

```
    uint8_t remove:1;
};
```

```
template <uint8_t size>
class SchedTable
{
    typedef Array<SchedEntry, size> Store;
public:
    typedef typename Store::Iterator Iterator;
    Iterator begin() { return table.begin(); }
    Iterator end() { return table.end(); }
    void add(Task *t, char id, uint16_t period, bool reschedule)
    {
        table.append(SchedEntry(t, id, period, reschedule));
    }
    void markRemove(Iterator i) { i->setRemove(); }
    void removeMarked();

private:
    Store table;
};
template <int size>
class Scheduler
{
public:
    //Scheduler();
    ~Scheduler() {}

    void dispatch();
    void loop();
    void add(Task *t, char id, uint16_t period, bool reschedule)
    {
        table.add(t, id, period, reschedule);
    }
    void tickUpdate();
private:
    SchedTable<size> table;
};

// implementation
template <uint8_t size>
void SchedTable<size>::removeMarked()
{
    uint8_t i=0;
    for (; i != table.size(); ++i)
    {
        if (table[i].getRemove())
        {
            for (uint8_t j = i+1; j < table.size(); ++j)
            {
                if (!table[j].getRemove())
                {
                    table[i++] = table[j];
                }
            }
        }
    }
    for (i = 0; i != table.size(); ++i)
    {
        if (table[i].getRemove()) break;
    }
    table.chop(i);
}
```

```
template <int size>
void Scheduler<size>::dispatch()
{
    for (typename SchedTable<size>::Iterator entry = table.begin();
         entry != table.end();
         ++entry)
    {
        if (entry->ready())
        {
            entry->run();     // synchronous
            entry->reset();

            if (!entry->periodic())
            {
                // one-time task
                table.markRemove(entry);
            }
        }
    }

    table.removeMarked();
}

template <int size>
inline void Scheduler<size>::loop()
{
    while (true)
    {
        tickUpdate();
        dispatch();

        cpu.goToSleep();      // save power
    }
}

template <int size>
void Scheduler<size>::tickUpdate()
{
    // check cooperative tasks
    for (typename SchedTable<size>::Iterator entry = table.begin();
         entry != table.end();
         ++entry)
    {
        entry->updateDelay();
    }
}

extern Scheduler<maxTasks> scheduler;

#endif /* SPSCHEDULE_HH_SEEN */
```

spschedule.cc

```
// spschedule.cc

#include "spschedule.hh"

SchedEntry::SchedEntry(Task *t, uint8_t taskId,
                       uint16_t period, bool reschedule)
    : tsk(t),
      periodTime(period),
      delay(period),
      id(taskId),
      again(reschedule),
      state(counting),
      remove(0)
{
}
```

uart.hh

```
// UART driver
// $Id: uart.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef UART_HH_SEEN
#define UART_HH_SEEN
#include <inttypes.h>

class Uart
{
public:
    Uart();
    ~Uart();
    bool putChar(uint8_t c);
    bool getChar(uint8_t &c);

    void reset();
private:
    enum { uartBaud = 19200 };
    uint8_t frameError:1;
    uint8_t overrun:1;
};

#endif /* UART_HH_SEEN */
```

uart.cc

```
// UART driver

#include "uart.hh"
#include <avr/io.h>
#include <util/delay.h>

inline void Uart::reset()
{
    UCR = 0;
    _delay_us(50);
    UBRR = (F_CPU / (16UL * uartBaud)) - 1;
    UCR = _BV(TXEN);
}

Uart::Uart()
    : frameError(0), overrun(0)
{
    reset();
}

Uart::~Uart()
{
    UCR = 0;
}

bool Uart::putChar(uint8_t c)
{
    if (USR & _BV(UDRE))
    {
        UDR = c;
        return true;
    }
    else
    {
        return false;
    }
}

bool Uart::getChar(uint8_t &c)
{
    uint8_t status = USR;
    if (status & _BV(RXC))
    {
        if (status & (_BV(FE) | _BV(DOR)))
        {
            if (status & (_BV(FE))) frameError = 1;
            if (status & (_BV(DOR))) overrun = 1;
        }
        c = UDR;
        return true;
    }
    return false;
}
```

leds.hh

```
// task to play some LEDs
// $Id: leds.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef LEDS_HH_SEEN
#define LEDS_HH_SEEN

#include <inttypes.h>

#include "spschedule.hh"

class LedTask : public Task
{
public:
    LedTask();

    void run();
private:
    volatile uint8_t curLed;
};

#endif /*  LEDS_HH_SEEN */
```

leds.cc

```
// task to play some LEDs
// $Id: leds.cc,v 1.1 2007/02/04 11:38:20 cvs Exp $
/* Note: LEDs on STK500 have inverse logic: 0=on, 1=off */

#include <inttypes.h>
#include <avr/io.h>

#include "leds.hh"

LedTask::LedTask()
    : curLed(2)
{
    // enable A1-A7 as output
    PORTA = _BV(PA1) | _BV(PA2) | _BV(PA3) | _BV(PA4)
        | _BV(PA5) | _BV(PA6) | _BV(PA7);
    DDRA |= _BV(PA2) | _BV(PA3) | _BV(PA4)
        | _BV(PA5) | _BV(PA6) | _BV(PA7);
}

void LedTask::run()
{
    uint8_t oldLed = curLed;

    if (++curLed == 8)
    {
        PORTA |= _BV(PA2) | _BV(PA3) | _BV(PA4)
              | _BV(PA5) | _BV(PA6) | _BV(PA7);
        curLed = 2;
    }

    PORTA &= ~_BV(curLed);
    PORTA |= _BV(oldLed);
}
```

io.hh

```
// simple I/O class system for small devices
// this is an asynchronous stream, i.e. output routines return before
// the output is actually finished
//
// $Id: io.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef IO_HH_SEEN_
#define IO_HH_SEEN_

#include "spschedule.hh"
#include "ring.hh"
#include "uart.hh"
#include <stdint.h>
#include <string.h>
#include <avr/io.h>

namespace
{
typedef RingBuffer<uint8_t, 32> BufType;
}

template <class Device>
class OutTask : public Task
{
public:
    OutTask(BufType *b) : buf(b) {}
    void run()
    {
        if (!buf->empty())
        {
            if (d.putChar(*buf->getNext()))
            {
                buf->popNext();
            }
        }
    }
private:
    Device d;
    BufType *buf;
};
```

```
static char endl[] = "\n\r";

class OStream
{
public:
    OStream() : transmitter(&tbuf), radix(10), overflow(0)
    {
        scheduler.add(&transmitter, 'U', 1, true);
    }
    ~OStream() {}

    OStream & operator<<(char const *s);
    OStream & operator<<(long i);

    void setRadix(uint8_t r) { radix = r; }

    bool ok() const { return overflow == 0; }
    void reset()
    {
        overflow = 0;
        tbuf.clear();
    }

private:
    BufType tbuf;
    OutTask<Uart> transmitter;
    uint8_t radix;
    char convBuf[12]; // sufficient for max signed long in decimal
    uint8_t overflow:1;
};

extern OStream out;

// implementation
// silently discards output that doesn't fit into the buffer
inline OStream & OStream::operator<<(char const *s)
{
    uint8_t const *us = reinterpret_cast<uint8_t const *>(s);
    if (tbuf.free() < strlen(s))
    {
        tbuf.append(us, tbuf.free());
        overflow = 1;
    }
    else
    {
        tbuf.append(us, strlen(s));
    }
    // the output task will do the rest
    return *this;
}
#endif /* IO_HH_SEEN_ */
```

io.cc

```
// simple I/O class system for small devices
// this is an asynchronous stream, i.e. output routines return before
// the output is actually finished
//
// $Id: io.cc,v 1.1 2007/02/04 11:38:20 cvs Exp $

#include "io.hh"
#include <stdlib.h>

OStream & OStream::operator<<(long i)
{
    ltoa(i, convBuf, radix);
    if (strlen(convBuf) > tbuf.free())
    {
        overflow = 1;
    }
    else
    {
        tbuf.append(reinterpret_cast<uint8_t const *>(convBuf),
                    strlen(convBuf));
    }
    // the output task will do the rest
    return *this;
}
```

test.hh

```
// task to test foreign interrupt timing
// $Id: inttest.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef TEST_HH_SEEN
#define TEST_HH_SEEN

#include <inttypes.h>

struct TestInfo
{
    TestInfo() : valid(0), testNo(0) {}

    uint8_t valid;
    int testNo;
    uint32_t value;
};
#endif /* TEST_HH_SEEN */
```

inttest.hh

```
// task to test foreign interrupt timing
// $Id: inttest.hh,v 1.1 2007/02/04 11:38:20 cvs Exp $

#ifndef INTTEST_HH_SEEN
#define INTTEST_HH_SEEN

#include <stdlib.h>

#include "spschedule.hh"
#include "test.hh"

class IntTest : public Task
{
public:
    IntTest();

    void run();

    TestInfo const &getInfo() const { return info; }
    void inc() { ++count; }
    void stopTest(uint8_t rest);

private:
    void startInt();
    TestInfo info;
    uint32_t count;
    uint8_t testing;
    int rcnt;
};

extern IntTest intTstTask;

#endif /* INTTEST_HH_SEEN */
```

inttest.cc

```
// task to test foreign interrupt timing
// $Id: inttest.cc,v 1.1 2007/02/04 11:38:20 cvs Exp $

#include "inttest.hh"
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "io.hh"

namespace
{
extern "C" ISR(INT0_vect)
{
    GIMSK &= ~_BV(INT0); // disable IRQ0
    PORTA |= _BV(PA0);  // stop LED0
#ifdef __AVR_ATmega8535__
    intTstTask.stopTest(TCNT2);
    TIMSK &= ~_BV(INT0); // disable counter2 overflow IRQ
#else
    intTstTask.stopTest(0);
#endif
}

#ifdef __AVR_ATmega8535__
extern "C" ISR(TIMER2_OVF_vect)
{
    intTstTask.inc();
}
#endif


} // unnamed namespace
```

```
IntTest::IntTest()
    : testing(0),
      rcnt(50)
{
    PORTA |= _BV(PA0);  // LED0 off
    DDRA |= _BV(PA0);  // PORTA0 as output
    PORTD &= ~_BV(PD2); // disable pull-up on INT line
    DDRD &= ~_BV(DDD2); // INT line as input
    GIMSK &= ~_BV(INT0); // disable IRQ0
    MCUCR |= _BV(ISC01);    // Interrupt 0 on falling edge
    MCUCR &= ~_BV(ISC00);
#ifdef __AVR_ATmega8535__
    TCCR2 = 0x1;    // full internal speed
    ASSR &= _BV(3); // internal clock
#endif
}


inline void IntTest::startInt()
{
    GIFR |= _BV(INTF0);  // clear any pending IRQ0
    GIMSK |= _BV(INT0);  // enable IRQ0
    PORTA &= ~_BV(PA0);  // light LED0
    count = 0;
#ifdef __AVR_ATmega8535__
    TIMSK |= _BV(TOIE2);// enable counter2 overflow interrupt
    TCNT2 = 0; // start counter2
#endif
}

// called from ISR
inline void IntTest::stopTest(uint8_t rest)
{
    count <<= 8;
    count += rest;
    testing = 0;
}

void IntTest::run()
{
    if (!testing)
    {
        if (rcnt == 0)
        { // test finished; update info and start new
            info.valid = true;
            ++info.testNo;
            info.value = count;
            rcnt = (rand() & 0xf) + 1; // we don't want 0
            rcnt <<= 10;    // makes roughly seconds
        }
        else if (--rcnt == 0)
        {
            testing = 1;
            startInt();
        }
    }
}
```

comm.hh

```
// task to communicate with host (output only)
#ifndef COMM_HH_SEEN
#define COMM_HH_SEEN
#include "spschedule.hh"

class CommTask : public Task
{
public:
    CommTask();

    void run();

private:
    volatile int lastIntTest;
};
#endif /*  COMM_HH_SEEN */
```

comm.cc

```
// task to communicate with host (output only)
#include "comm.hh"
#include "io.hh"
#include "inttest.hh"

namespace
{
void check()
{
    if (!out.ok())
    {
        out.reset();
        out << "ERR" << "OUT" << endl;
    }
    if (cpu.overflowed())
    {
        out << "ERR" << "CPU" << ": " << cpu.overflowed() << endl;
    }
}
}

CommTask::CommTask()
 : lastIntTest(0)
{
    out << "Start Tests" << endl;
}

void CommTask::run()
{
    check();

    TestInfo info = intTstTask.getInfo();
    if (info.valid && (info.testNo > lastIntTest))
    {
        out << "INT" << ": " << info.testNo << ": " << info.value << endl;
        lastIntTest = info.testNo;
    }
}
```

main.cc

```
// setup tasks and scheduler and start

#include "spschedule.hh"
#include "leds.hh"
#include "inttest.hh"
#include "io.hh"
#include "comm.hh"

Scheduler<5> scheduler;

OStream out;

IntTest intTstTask;

LedTask leds;

CommTask comm;

int main()
{
    scheduler.add(&leds, 'L', 1000, true);
    scheduler.add(&intTstTask, 'I', 1, true);
    scheduler.add(&comm, 100, 'C', true);

    scheduler.loop();

    return 0;
}
```