Small C++

C vs. C++ code size on 8-bit AVR

Detlef Vollmann

vollmann engineering gmbh, Luzern, Switzerland

LUGS, Zurich, January 2014

Overview

Introduction

- **Hardware**
- Design
- C++

Code

Hardware Patterns

- 8bit Coprocessor
- Realtime Coprocessor
- Peripheral Coprocessor
- Hardware Watchdog

8bit Coprocessor

- Name: "8-bit Coprocessor"
- Problem: Some logic is hard to do in hardware
- Forces:
  - some things are hard in hardware
  - but don't fit into main CPU
  - 8-bit processors a cheap
    - as components and in manufacturing
- Solution:
  - Use separate 8-bit coprocessors

Realtime Coprocessor

- Name: "Realtime Coprocessor"
- Problem:
  - Favoured OS clashes with realtime requirements
- Forces:
  - Some protocols or hardware have hard realtime requirements
  - Selected well-known widely-used OS cannot provide the required realtime guarantees
- Solution: Add a separate, dedicated realtime controller
- Consequences:
  - additional hardware costs
  - devide and conquer

Peripheral Coprocessor

- Name: "Peripheral Coprocessor"
  - variation of "Realtime Coprocessor"
- Problem:
  - Special protocol with hard latency or throuput requirements
- Forces:
  - similar as for "Realtime Coprocessor"
  - implementation in main processor would be problematic
  - same protocol is used in different systems
  - doesn't exist as COTS
- Solution:
  - implement it in software on a separate processor

Hardware Watchdog

- Name: "Intelligent External Watchdog"
- Problem: Watchdog timer devices are inflexible
- Forces:
  - external watchdog shall reset the main processor after some time of missing heartbeat
  - but boot time is longer
  - and not during firmware update
  - after specific number of unsuccessful attempts some alarm shall go off
- Solution:
  - use separate "8-bit Controller" with watchdog software

AVR

AVR is a popular 8-bit microcontroller architecture by Atmel

- tinyAVR start at 512B flash and no RAM (but 32 registers)
- megaAVR start at 4K flash and 512B RAM

AVR is used on the Arduino boards

Overview

Introduction

- Hardware
- **Design**
- C++

Code

Flexible Design

"Design for Change"

Keep the design flexibel

- extendable:
  - It's easy to add new functionality
- adaptable:
  - It's easy to change existing functionality
- reusable:
  - Reuse of parts in other systems
  - Reuse parts from other systems

Object Benefits

- Reliability
  - It runs, and runs, and runs ...
  - smaller units
  - cleaner code
  - more robust code
- Reusability
  - Special versions, different hardware and similar systems
  - classes as re-usable unit

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.

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

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.

Overview

Introduction

- Hardware
- Design
- **C++**

Code

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

Overview

Introduction

- Hardware
- Design
- C++

Code

AVR C++

- GCC has an AVR backend
  - No tinyAVR
  - RAM sizes starting from 128B (old devices)
- So GCC C++ also works
  - No exceptions
  - No placement new
  - No virtual destructors
  - No Standard C++ library
- AVR Libc library (http://www.nongnu.org/avr-libc/)
  - Provides fairly complete C library
  - Even `<stdio.h>` and `malloc()`
- Arduino provides a C++ library that's not used here.

Code

- “Hello, World!” embedded:
  - blinking LEDs

Questions

- ???????????????????????????????????