Code Sharing using C++ between Desktop Applications and Real-time Embedded Platforms

Overview

The general product was a data collection and messaging system that could display real-time business data on the PC desktop running Microsoft Windows or on LED “message boards” which contained custom embedded CPU boards.

- Description of Software and Hardware Environment
- Development Issues and Concerns
- Lessons Learned

Hardware and Software Environment

- **Pentium-class PCs for desktop systems**
  - Running Windows 95/98/NT in a single-threaded, multi-processing fashion
  - Using Microsoft C++ compiler and debugging tools

- **Custom CPU board based on a 68340 or PowerPC 860 (601 processor core) running at 25 MHz**
  - Pre-emptive real time operating system
  - Diab Data C++ cross-compiler & SDS debugger under Windows NT

- **Software and Programmer Descriptions**
  - ~75 KLOC of shared C++ that included limited templates, RTTI support due to multiple inheritance schemes, and no C++ exceptions
  - ~10 KLOC of C++ and C code specifically for embedded target support which included RTOS task implementations and low-level drivers for hardware support
  - Approximately 12 programmers with a 2/3 – 1/3 split among software and firmware engineers

- **Code Development Process**
  - Classes were developed and unit tested on the desktop
  - Classes then compiled and unit tests ran on embedded target
  - Classes fit into RTOS framework for the embedded application
  - System integration tests:
    - System tests of PC applications were run in parallel with the system tests of the embedded application that used the shared code
    - A full system integration test that included PC applications and embedded application was then run

Development Issues

- **Resources**
  - Avoid duplication of effort by having 1 programmer write for 2 targets instead of 1 programmer per platform
  - Code executes exactly the same on both platforms
  - Desktops have better tool sets for development, debugging, profiling, memory and performance analysis
  - Development can proceed on the desktop if the embedded target is not available

- **Code Guidelines**

  Guidelines need to be established for acceptable behavior that include performance in speed and size, maintainability, etc. Areas to consider include:

  - C++ Templates
  - C++ Exceptions – Throw/try/catch sequences introduce performance and size hits which may be undesirable
  - Portable, standard coding techniques should be used. Avoid pointer tricks and assumptions or reliance on data word sizes, byte-ordering, etc.
  - Minimize non-portable, target-specific code which must be identified and isolated either in separate modules or with the use of **`#ifdef/#elif/#endif`** bracketing.
  - Utilize interface definitions with virtual functions and abstract classes to abstract processor, operating system, or other target-specific dependencies

- **Architecture Issues**

  Differences in software and hardware architectures between the different platforms must be identified and isolated in the code. Some of these areas include:

  - Processor-dependent
    - Endian differences – important for communication between different endian-based computers
    - Compiler-introduced “optimizations” which include aggregate data structure padding, differing sizes for data types
    - Computer resource availability
    - Raw processor speed – 300+ MHz Pentiums hide a lot of problems that arise on a 25 MHz 68xxx processor.

  - Software
    - Single-threaded vs. multi-threaded environment
    - Real-time systems have deterministic time constraints vs. run-to-completion environment on the desktop

- **General Issues**

  Many of the issues that arise in a shared code development environment are not rocket science, but common sense:

  - Each developer must be responsible for maintaining his/her code in both environments
  - Each developer must be willing to adapt his/her code to fit into the constraints of the embedded target
  - Much of successful code sharing relies on sensible human relations:
    - Open and constructive communication
    - “Diplomacy”
    - Ego-less perspective

Lessons Learned

- **Software Development Lessons**
  - Design and develop for both target environments simultaneously
  - Code sharing enhanced robustness and reliability
  - Code sharing saves a lot of effort
  - Performance can be a big issue
  - Someone must be responsible for creating and enforcing standards by which all shared code authors must follow

- **Tools**
  - Know your compilers!
  - To a lesser degree, know your linker/loader
  - Utilize the rich off-the-shelf tool set available for desktop computers, then use the specific tools for the embedded target either off-the-shelf or roll-your-own tools