HARDCORE MICROCONTROLLER PROGRAMMING

manuel odendahl - wesen - wesen@ruinwesen.com - http://ruinwesen.com

Introduction

This article compiles some of the programming and debugging techniques discovered over the last year while writing the Mididuino framework, the code environment running on our MidiCommand MIDI controller. Despite the apparent simplicity of the device (8-bit microcontroller, 4 buttons, 4 rotary encoders, a simple character-based text-display, and 2 MIDI ports), building a GUI and MIDI framework turned out to be pretty involved, due to the restricted computational resources and the inherent difficulties of embedded programming. To alleviate some of the difficulties, a few techniques of agile programming were adapted to work in an embedded environment, including unit testing, rapid prototyping, and automatic building and deployment. C++ was used as the core language instead of C, in order to increase programming effectiveness. This turned out to be a good choice, but came at the cost of increased program size. Thus, a big part of optimizing was spent on reducing program and data size. The realtime nature of musical processing also led to an interesting journey into the intricate world of low-latency and timing-dependent programming. Finally, debugging a hardware devices is quite different from debugging software running on a desktop computer, and led to the creation of useful embedded software tools. Each of these topics is so vast that a whole article (if not book) could be written about each of them. This article is meant more as a collection of thoughts and pointers rather than a detailed technical article. For the technically inclined reader, the complete source code, schematics and tools described in this article, as well as a collection of development-related blog posts are available on our website at http://ruinwesen.com/ .

The Minicommand MIDI controller features a small 8-bit microcontroller (the Atmega64), featuring 64 kB of code space (used to store programs and static data) and 4 kB of RAM (with an additional 64 kB of RAM added externally). It features 2 MIDI interfaces: 2 inputs and 1 output. MIDI is a serial protocol used control synthesizer and other musical devices. A MIDI controller allows the musician to control parameters of a synthesizer, trigger notes, and synchronize the tempo of different sequencers. There are a number of different musical messages that can be transmitted over MIDI: notes, control values, tempo sync as well as generic binary data. These messages immediately trigger a musical response, and thus have pretty hard real-time requirements. Even slightest variations on the order a few hundred microseconds can lead to phasing and other noticeable sonic changes.

The Mididuino framework is completely opensource. It is split up in individually usable "libraries", featuring common data structures and tools, a MIDI parser and various MIDI utilities, a GUI library used to create user interfaces on the MiniCommand, as well as elaborate libraries used to control specific synthesizers (such as the Elektron MachineDrum and Elektron MonoMachine). Besides being an elaborate MIDI controller, it focuses on providing a practical development environment for users interested in exploring their own musical concepts.

Agile Microcontroller programming

Agile programming is a broad term used to describe a number of programming and organizational techniques. It emphasizes working with code as an ever-changing medium, rather than trying to enforce more rigid traditional methodologies. Most embedded software is not often updated: agile programming is not very common in traditional embedded software development. However, with the advent of self-programmable microcontrollers , shorter release cycles and a more flexible approach to firmware development has become available.

The actual philosophy of the framework was heavily influenced by the Arduino platform. As such, Arduino is a pretty simple microcontroller development board (featuring a smaller version of the Atmega-processor used in the Minicommand), which is programmed by writing short firmwares (called "sketches") in a rather simple development environment. Most of the actual complexity of setting up and accessing hardware resources is hidden behind a simple C++ library. The actual beauty of the Arduino environment doesn't lie in its technical realization, but rather in the way it completely changes the approach to building embedded software: rather than setting up a Makefile, writing the hardware setup code and transferring the firmware using a separate hardware programming, in Arduino you write a few lines and press the "Upload" button. This simplified setup makes a huge difference when it comes to trying out new ideas. The Arduino editor was adapted to work with the MidiDuino framework, and is available to every user who wants to develop his own software.

This whole approach is made possible by a small program called the bootloader. On the Atmega microcontroller, this program resides at the higher-end of the program address space, and is executed right after powering on. The bootloader uses one of the available communication channels or storage resources to reprogram the flash memory storing the program code. On the Arduino, this is done over the serial USB interface, on the MiniCommand, this is done using the MIDI port. This avoids the use of dedicated programming tools, and also allows non-technical users to update the firmware, using a software called the "Patch Manager". On startup, the software connects to the manufacturer website, downloads a list of new firmwares along with documentation, and allows the user to choose a new firmware for his device. It is a small opensource app store for embedded devices.

Using a source version control system (VCS) has become commonplace in the software industry. For embedded programming, the toolchain used to produce the actual binary firmwares is of crucial importance. It is thus quite practical to use the VCS itself to store the toolchain along with the sourcecode, using it as a lightweight configuration management system. The quick branching and merging offered by tools such as git (used for the Mididuino framework) makes it possible to fix problems or implement new features in parts of the system, while still allowing full access to older builds. This is very important because of the intricate interplay that embedded software can have with the actual hardware. Even a few additional opcodes can change timing in such a way that a new bug arises. Every firmware shipped is automatically tagged, along with the used tools and toolchain, to simplify maintenance.

Unit testing is an important part of agile development. Software is tested and exercised by a number of small programs called "unit tests". These are executed by the programmer to make sure that changes haven't introduced new bugs. Unit tests are most useful when they can be run frequently and quickly: most of the framework was split into a small hardware dependent part, and a bigger platform independent part containing most of the functionality. Host-platform abstractions for most of the hardware features of the embedded platform are provided (MIDI ports, timing and GUI). The unit tests are run on the host platform, and test most of the features of the framework in a few seconds. This is especially crucial to make sure that the the low-footprint data structures and data parsers are working correctly. The timing dependent part of the framework has been designed to allow for the testing of timing-dependent features (such as testing the low-latency requirements of the MIDI interface).

With the availability of over a dozen of different firmwares in the PatchManager and the growth of the framework to over 30,000 lines of code, the code base grew slightly out of control. To manage the complexity, a number of custom "tools" (actually crude perl and shell scripts) were written to automate builds. The scripts run the unit test suite, build every available firmware (testing the API compatibility of firmwares and the code and ram requirements) and upload updated versions to the website. Each released version is tagged in the version control, and the documentation for the framework and the individual firmwares is automatically generated. The platform part of the framework (some MIDI utilities, the PatchManager and the MidiDuino editor) are built on different OS using the buildbot system. It also allows to have a plot of the evolution of firmware size over time.

C++ on a Microcontroller

The first firmwares for our MIDI controllers were written in the "traditional" languages C and assembler, making for small memory footprints and a manageable amount of portability. However, with the introduction of the rapid prototyping techniques described in the first part of this article, the decision was made to switch to the C++ programming language. C++ is a pretty convoluted "multi-paradigm" programming language: supporting standard structured programming, object-oriented programming as well as metaprogramming. It is one of the few languages giving the programmer absolute control over memory allocation and placement and functions-call overhead, which is crucial on a restricted environment such as the Atmega microcontroller. The "virtual" keyword, which is necessary to have method overloading (and introduces a further indirection on method calls by storing per class function pointers), may seem slightly unnecessary on the powerful computers we have nowadays. However, one can save up to hundreds of bytes of program size when using them only when necessary (a few bytes for the storage of the pointer, and more importantly the removal of function pointer lookup code on function calls). C++ is a huge language, and the only way a developer can tackle is to use a narrowly defined subset. We banned the use of dynamic memory allocation, while allowing virtual methods and multiple inheritance (mostly to provide callback type checking). Templates are used in very specific cases to create type-generic data structures, but are typically avoided because they make program size grow drastically.

The main reason for chosing C++ was the creation of an object-oriented GUI framework. While theoretically possible in C, using function pointers and class-like structures, it just is much easier to write and debug such a class hierarchy (containers, event handlers and callbacks) in an object-oriented language. Furthermore, classes for typical hardware features such as UART (serial interface for MIDI), SPI bus (to access the SD-card storage), display or PWM-modulation for LED intensity can be split up into a general parent class grouping hardware-independent functionality and a lower level child class implementing the actual hardware interface. This is usually the reason for a lot of macro "#define" and "#ifdef" magic in traditional C firmwares which can be more intuitively modeled by OO-programming. Templates are used for type-generic low-level data structures such as ring buffer (byte ring-buffers, short ring-buffers, pointer ring-buffers), vectors, list (with statically allocated memory pools) and callback dispatchers. Actually, the assembly code generated by the ring-buffer template is comparable in efficiency with hand-written assembly code.

The main disadvantage of C++ are the increase in code size. This is mainly due to the pointer arithmetic required to calculate member variable offsets and function pointers on method calls, as well as the lacking optimization for small 8-bit platforms (this is being worked on by the GCC maintainers). Templates also lead to a big increase because of code duplication. Another reason for the code size increase is the rather "primitive" toolchain support. For example, avr-g++ always generates and includes constructor methods, even if those just statically initialize member variables. The work done on link-time optimization and static program analysis, which is well underway in GCC itself and in new compiler projects such as LLVM, will probably resolve a lot of these issues. The actual overhead of C++ should actually not be much more than that of C with carefully crafted code.

Optimization

Optimization is an important topic in embedded development, due to the restrictions of the platform. Speed issues are usually not critical, despite the 16 Mhz of the Atmega microcontroller. However, it is useful to know that most mathematical functions (especially when working with floats, as often is the case on the Arduino platform) take a long time to execute. The microcontroller works with 8 bit values (with a few operations like addition and subtraction being possible on 16 bit values), so that arithmetic with bigger data types such 16 bit and 32 bit integers leads to a noticeable computational overhead (a 16-bit addition requires about 3 times as much instructions as an 8-bit addition, which means that it is approximately 3 times as slow and takes up 3 times as much code space).

A profiler is a useful tool to look for performance-related bottlenecks. The MidiDuino framework includes a rudimentary profiler that sends the current instruction pointer position at regular intervals. It has a small impact on the performance of the system, but is very useful to determine where the CPU bottlenecks are. These bottlenecks are usually quite different than when running the code on the host platform.

The real constraint however is memory size, both volatile memory (RAM for data storage) and static memory (flash for code storage). A big amount of code size can be saved by the judicious use of compiler flags. The standard flags (after careful selection) used by the MidiDuino framework are: -Os : use size-optimization, -ffunction-sections -fdata-sections -Wl,--gc-sections : store each function and each variable in its own section so that the linker can to dead-tree elimination, -Wl,--relax : use linker relaxation (use relative calls when functions are nearby), -fno-exceptions -fno-rtti : don't use C++ exceptions nor RTTI, -mcall-prologues : pack function prologues and epilogues into functions to avoid code duplication, -fshort-enums : use the smallest integer size for enum values, -fpack-struct : pack structures, -fno-default-inline -fno-implement-inlines --param inline-call-cost=2 -finline-limit=3 -fno-inline-small-functions : finetune the inlining heuristics, -fno-tree-scev-cprop : vectorize the outer loop for size optimization, -fsplit-wide-types : allocate the different registers of a wide data type separatly, instead of keeping them in neighboring registers, -fno-threadsafe-statics : avoid the use of threadsafe code to access static variables.

Programming practices have a tremendous effect on code size and memory usage. Big switch statements can be avoided by rewriting them to use a function pointer lookup table. Binary parsers can be constructed to work on an incoming stream of tokens rather than on a byte buffer. Static data can be moved from volatile memory to program memory. Extracting commonly used functionality to individual functions, as well as creating functions providing often used default arguments can reduce code size as well. When using arrays, structures and especially arrays of structures, it is often useful to reorganize loops (for example to have the iteration pointer stored in a register) so that the compiler doesn't produce too much dereferencing code and memory indexing. The Atmega64 processor has 3 pairs of registers that can be used for indirect indexed access, with one usually pointing to the current object, so that one has to be careful not to do too much parallel access on arrays. Small tricks such as using predecrement statements (--i instead of i--) can save up a few bytes as well by making it easier for the compiler to use the underlying hardware instructions. A good resource for small memory programming practices is the freely available book "Small Memory Software" at http://www.cix.co.uk/~smallmemory/ .

Toolchains utilities are available to profile the code size of individual functions, data structures and files. These utilities are avr-size, which prints the size of individual ELF sections, and avr-nm --size-sort, which prints the size of individual symbols. The output of these tools can be interpreted by a perl script to pinpoint which modules are most in need of size optimization. While assembly programming as such is not usually necessary, it is very useful to be able to read the assembler output of the compiler to understand how certain programming language features are translated. One can use the avr-gcc -S -fverbose-asm command to generate the assembler output of a C file, and use the avr-objdump -S command to disassemble a linked firmware binary. In order to get comfortable with the process, it is quite useful to write small dummy programs of a few line to see how a division is compiled, how a class access is compiled, how a virtual method call is compiled, etc… This knowledge is invaluable while debugging.

Latency and Timing

Latency and timing are critical issues in an embedded system, as it must interact with external chips and real-world hardware. Constraints can be hard (they must be met), or soft (they need to be met as quickly as possible, but there is room for variations). Some constraints are set by the embedded application: on the MIDI controller, we have pretty soft timing requirements when interfacing to the shift registers used to read out the buttons and encoders, and much harder requirements when it comes to MIDI latency. These parallel requirements make the use of some kind of code concurrency necessary. On the desktop computer, one would use different processes. An important design consideration is whether to use an OS (often called RTOS in the embedded world because of the functionality focusing on real-time processing) or not. The overhead of an OS, both in code size, code complexity and in memory usage, can be quite drastic, as each execution thread needs its own stack, its own registers and further associated data structures. On an 8-bit controller, which often runs simpler applications, the concurrency requirements can be dealt with with interrupt routines.

The MidiDuino framework uses a runloop handling the background tasks and the uncritical parts of the software, while interrupts are used to react to external hardware (by using interrupt on pin changes for example, or having interrupts triggered by received or sent bytes on a communication bus). Furthermore, one can use the internal timers of the microcontroller (the Atmega64 has 3 timers) to schedule regular operations. The runloop parses incoming MIDI data, reacts to user events, and generally does the work required by the MIDI controller firmware. MIDI communication itself is handled by an interrupt routine, so that sending MIDI data out of the runloop is non-blocking. Incoming MIDI bytes are stored in a ring-buffer and processed later on, outside the interrupt (of course, these ring-buffers need to be big enough to avoid overflows). Furthermore, the incoming MIDI interrupt directly handles MIDI real-time messages by passing them to the MidiClock subsystem. Regular tasks are scheduled to be run in the mainloop, but this is nowhere near deterministic. Scheduling MIDI events is done through the MidiClock subsystem, which uses one of the timer interrupts to run an accurate internal clock.

The IO timing requirements are twofold: the first is the generation of a clock signal for the shift registers, and the second is reading out the buttons and encoders fast enough in order not to miss an action of the user. A second timer interrupt is used to run the IO reading code at about 2 khz. The clock is generated with interrupts turned off, so that the timing is deterministic, and the incoming IO events are stored in an event queue to be further processed in the actual soft-realtime runloop. The user of the framework has the possibility to add his own code to the IO polling interrupt subroutine in case some hard real-time action needs to be taken upon user action. The communication with the SD-card, which is done through SPI, uses the hardware SPI capability of the Atmega64, and can transfer big chunks of data without impacting the overall performance of the firmware.

The MIDI controller can be slaved to an external MIDI clock, and has to run the internal sequencers on this clock. The master clock sends a MIDI clock tick (byte 0xF8) at regular intervals, every 96th note. The firmware can either directly update its internal sequencer counters and react on that incoming message. This is very simple to implemented, but adds a fixed latency for receiving the byte and interpreting it, so that chaining multiple devices will lead to a noticeable delay between the first device in the chain and the last. The other possibility, which is quite tricky, is to use an internal digital phase locked-loop that synchronizes itself to the external events. This digital PLL is implemented by using an internal clock (updated by a timer interrupt), calculating the phase difference between the internal oscillator and the external events, running the difference through a low-pass filter, and updating the frequency of the internal oscillator. While this is quite simple to implement with analogue components, on a microcontroller, jitter, interrupt latency and especially numerical precision start to become an issue. To have enough precision to achieve a wide range of musical tempos (from 20 bpm to about 300 bpm), a 32-bit clock counter needs to be used. Calculating the low-pass on every phase difference then starts a noticeable overhead, so that the low-pass calculation is offloaded to the main loop, and executed when the actual timer interrupt routine sets a flag. This technique of "splitting" interrupts into a critical real-time part and a more relaxed soft real-time part is called "bottom half / top half".

Debugging Techniques

The main hurdle when developing for embedded systems is debugging. Bugs can creep in pretty easily due to the restricted resources (it is for example quite easy to run out of stack space) and the concurrency problems created by interrupts. A number of bugs can come from the hardware itself: the device resetting itself spuriously due to a short or a bad solder connection, the power supply not being within specs, a faulty external chip or even grounding problems. It is also much more common for the toolchain to have bugs. One version of GCC for example didn't correctly save registers on interrupt calls, leading to hard to trace bugs when using the IO. It is often quite useful to go through the assembly listing by hand, questioning everything the compiler does. All in all, the programmer thus has to be much more creative while debugging, keeping an eye on all the layers of the system, trying to assess if the bug is caused by the hardware or the software.

A number of tools can assist him in his search. The most valuable tool is probably the oscilloscope, used to examine voltages, search for shorts, trace serial communication or look for the interplay of various parts of the circuit (on a multichannel scope). Knowing how the scope works is probably as important as knowing how the compiler or debugger works. Using triggers, saving traces in the scope's memory, and the various mathematical and measuring tools (FFT, pulse-length measuring) can save a huge amount of time while looking for a problem. To have easy access to various signals on the boards, small testpoints are very useful. It is possible to use the scripting possibilities of the EAGLE layout tool to generate a printout of the testpoints, their location on the board and their signal. A similar tool is the logic analyzer, which is basically a multichannel digital scope which can be used to trace data communications. Most logic analyzers come with a software-counterpart running on the host system that is able to analyze the recorded data, and decode various protocols. It can be a huge timesaver when debugging UART or SPI communications.

One can also achieve a lot with simpler solutions. Toggling LEDs at certain points in the code allows to have a direct visual feedback, and do post-mortem analyses if the system crashes. One can also use piezos instead of LEDs to have an auditive feedback. It can be quite useful for example when debugging timing events, as the ear is much more sensitive to subtle variations than the eye. A regular interrupts will change in pitch if it misses calls, and slight variations in latency on simultaneous events will result in phasing clicks. A further advantage of using piezos is that the programmer doesn't have to keep an eye on LEDs or the scope to hear bugs. All these techniques have the advantage that they are non-intrusive: they don't modify the running software in any way, although in some cases using the scope or piezos will have an effect on the electronics.

When one decides to modify the running software, which can be difficult when searching for timing related problems, one can use the communication buses or the connected hardware to have an output channel. For example, the MidiDuino framework allows printf statements to be sent over the MIDI interface, and then captured and displayed on the host side. The display can also be used to display short debugging information. However, one has to keep in mind that not only does executing the debugging code change the timing behaviour, it will also affect the internal state of the device (putting data in the communication ring buffers, copying data to memory, etc…). Thus, this kind of debugging technique is mostly useful to look for high-level programming errors which can also be looked after on the host (and in the unit tests). On the MIDI controller, a debugging technique similar to the piezo approach is sending actual MIDI note and MIDI data to a syntheziser. One can for example observe the overall load of the system, or another metric by controlling the pitch of a note. Sending a note trigger at the start of each function also creates an interesting way to debug calls by comparing the melodies created when running a piece of software.

Finally, a very useful approach to debugging microcontrollers is using the embedded debugging facilities, usually accessible over JTAG or in the case of the Atmel microcontrollers, PDI or debugWire. An adapter is connected to the host device, and a debugger on the desktop computer can control the execution of the firmware on the microcontroller. The Atmega64 controller allows for 5 breakpoints, and data can be read and written over JTAG. However, the standard gdb debugger is not very efficient to use, as setting breakpoints and singlestepping through the code can be annoyingly slow. We started working on a Javascript bridge to JTAG, writing the actual debugger in Javascript. This allows the programmer to "customize" the debugger depending on the problem it is looking to solve. The Javascript debugger has full access to the debugging information stored in the binary file, and can for example log the content of a data structure on each breakpoint pretty easily. Using an external debugger however has even more drastic timing consequences than using the internal communication capabilities, and is often slower than using other debugging tools.

An important part of building reliable embedded systems is making sure they will withstand a certain amount of physical abuse. Always try to break the system to know where that boundary is. Without investing in high-end testing instruments, it is quite easy to stress the system for different criterias. Temperature stress can be provided by putting the device in the oven at different temperatures, putting it in the fridge or in the freezer, and then quickly moving it from oven to freezer. This will put a high amount of thermal stress on the components and on the assembly as well. Of course, one can try to kick, shake and rattle the device itself, especially if the construction involves mechanical parts. Vibration testing can be simulated very roughly by using various household devices, such as a dishwasher, a car, a subwoofer or a drilling gun. High voltage spikes can be easily created by using a piezo out of an electronic firelighter. Zapping the device at the various points of contact with the user can show if the device withstands static electricity. Most trouble on the MiniCommand involved the chinese display, and various corrective measures had to be taken after this "cheap" testing. In the case of the MIDI controller, the actual environment in which it is going to be used is quite hostile: a danceclub at 3 in the morning, full with lights, sweat, loud music and unpredictable music. It actually is a good situation for stress testing any kind of electronics, provided you can take it into the club.

Stress testing on the software level can be done by using fuzzying, which is feeding the device a huge number of random or semirandom data to see if parsers and other parts of the code can handle them. In the case of communication interfaces, it is necessary to test the code under full communication load to see if the device is fast enough to handle such spikes. It is also quite useful to crank up the frequency of timing related events or to introduce deliberate delays to see at what moment the concurrency starts to break down. In the case of stored data (on the SD-Card, or in the eeprom), a good test is to fill it with spurious random data to see if the device can handle it.

Conclusion

This article was a very short overview of the different approaches and techniques involved in writing software for embedded devices. Having a deep and detailed knowledge of the whole system, starting from programming language, to compiler internals, to knowledge of the hardware involved and especially a good affinity with the development tools (software and hardware) is key to writing solid embedded software (maybe even more so than development for desktop computers). It is also very important to keep an open mind and to often think out of the box, as bugs can hide in a completely different part of the system, and can not be as easily isolated as in traditional software.