Educating Embedded Systems Hackers

A practitioner's perspective

Sagar Behere
Kungliga Tekniska Högskolan
Brinellvägen 85
Stockholm, Sweden
behere@kth.se

Martin Törngren
Kungliga Tekniska Högskolan
Brinellvägen 85
Stockholm, Sweden
martint@kth.se

ABSTRACT

Theoretical skills imparted during university education in Embedded Systems often surpass their practical counterpart. The contribution of this paper is a defined set of practical skills which bridge the gap between a sound theoretical education in embedded systems and the skillset acquired by experienced practitioners in the field. The presentation of each skill is accompanied by common solution patterns, state-of-practice technologies, and a set of exercises to provide practical uptake of each skill. The proposed skillset is based on consistent observations over the years, of graduating students performing "hands-on" projects; the proposed approach for imparting the skillset is motivated by experiences with Embedded Systems education at The Royal Institute of Technology (KTH) in Sweden.

Keywords

Cyber-Physical Systems, Realtime, Data logging, Communication, education

1. INTRODUCTION

The term *hacker* traditionally[38] refers to "A person who delights in having an intimate understanding of the internal workings of a system, computers and computer networks in particular." The term is used in a more general sense in this paper, to refer to a person who complements good theoretical knowledge of embedded systems with a strong competence in the practical aspects of constructing and programming those systems. Such a person would be ideally suited for specifying the overall technical architectures for complex embedded systems, selecting appropriate hardware and software stacks for different parts of the system, coordinating and integrating output from individual contributors (code, models, algorithms etc.) and in general, bootstrapping and nurturing the development of a complex embedded system. As a norm, hacking pushes the boundaries of what is practically possible and supplements good engineering, which deals with applying known knowledge within a set of constraints (economic budget, safety, etc.). The embedded systems hacker, in our portrayal, is differentiated from domain specific technical specialists by having the ability to make and execute good system level technical choices. In most situations the hacker would prioritize the practical ability to create, while retaining a desire to eventually explore the theoretical intricacies.

Embedded systems hackers as described above are formed today, more often than not, by dint of their own enthusiasm and curiosity. Their practical technical skills are acquired and honed through multiple projects and each individual creates (and hopefully updates) his/her own toolbox of technological solutions and solution patterns which are then repeatedly applied to subsequent projects. In every classroom, there are some motivated students who go beyond the curriculum and tinker and play and teach themselves the tools and technologies in vogue. They go on to become natural technical leaders of their projects, and are appreciated in industry and academia alike. As educators, it is our duty to stimulate more students to fall into this category.

Therefore, it is worthwhile for embedded systems educators to inquire into the practical skills and familiarity with technological solutions that should be imparted via academic curricula in order to facilitate the creation of such hackers. This paper suggests a set of such skills and technological solutions. The suggestions are based on a decade of experience in building complex embedded systems (e.g. the Scoop project[52]), discussions and investigation with industry and academic representatives including in the context of developing an agenda for Cyber-Physical Systems[12, 17], as well as experiences in embedded and mechatronics education[51, 54], and observations of students who are nearing the completion of their academic degrees and involved in research projects. The presentation of each skill in this paper is accompanied by some relevant solution patterns, current technological solutions, and exercises to hone the skill under consideration. The technological solutions mentioned are based on state-of-the-art, popular, open source software and hardware. The target audience is educators developing curricula for embedded systems education as well as aspiring embedded systems hackers looking to expand their skills and making themselves more marketable.

The focus of this paper is on embedded systems hardware and software aspects, especially during the early prototyping phase of product development.

WESE2014 October 16, 2014, New Delhi, India

2. THE TECHNICAL SKILLS SET

2.1 Platforms and functionality distribution

2.1.1 Skill

The term *platform* as it is used here, refers to the hardware and software stack(s) in the embedded system. Platforms typically range from minimal 8-bit microcontrollers which are programmed on "bare metal" to more capable 32 (and sometimes 64) bit Single Board Computers (SBCs) which run standalone, full-fledged desktop operating systems. Examples of the latter include ARM® based devices capable of running a complete desktop Linux distribution. Even for low end 8- and 16- bit microcontrollers, there is a rising trend of using tiny operating systems bundled into the application code. Examples of such operating systems are FreeRTOS™[19], Erika Enterprise[18].

A good engineer knows when to use a single high end microcontroller for all needed functionality and when to split up the functionality among different platforms with varying hw+sw stacks. The practical tradeoffs associated with this decision are sometimes not directly visible. For example, it is perfectly feasible to use a Linux process to toggle a GPIO output of a high end microcontroller and generate a stable PWM signal. A good engineer knows this, but also knows that it is just vastly simpler to connect a cheap 8 bit microcontroller to the Linux platform and use its built-in PWM outputs to do so. (In this case, because there is an effort associated with patching the Linux kernel to make it hard real-time, possibly writing a hard realtime device driver for the gpio subsystem and balancing task priorities, interrupt handlers, concurrency and reentrancy aspects to ensure a stable software generated PWM waveform.)

Educators need to ensure that students are exposed to platforms of varying capabilities and that the students are sufficiently proficient with them to make wise decisions regarding their usage. A minimum set of such platforms would be a tiny 8-bit platform without an operating system, a 16-bit platform with an operating system that is linked into the application and a 32 bit platform running a standalone operating system into which application programs can be loaded and executed.

The advent of multicore chips and FPGA technology has changed the game somewhat. With multicore chips, it is possible, in principal, to group tasks with specific, shared characteristics into sets and allocate a core to each task set. This approach then requires due consideration of multicore related issues, and appropriate capabilities from the operating systems/hypervisors. FPGAs blur the distinction between hardware and software and their usage can result in inherently realtime functions, efficiency gains for compute intensive functionality as well as optimum functionality distribution.

2.1.2 Solution pattern

A generalized solution pattern is shown in Figure 1.

The pattern includes a high-end microcontroller with a standalone, non-realtime operating system(the "Fat Stack"), a low end microcontroller for low-level input/output and a third microcontroller devoted exclusively for realtime tasks. Compute intensive tasks may be allocated among the microcontrollers based on their realtime characteristics and resource requirements. Depending on intended application, one or more of the microcontrollers may be eliminated and/or their functionality may be combined into a single microcontroller.

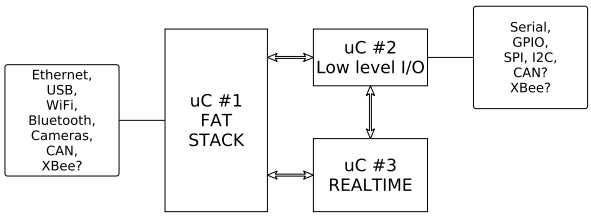

Figure 1: A generalized architecture pattern

While an extensive defense of the pattern is beyond the scope of this paper, we will return to this pattern repeatedly in the subsequent sections. For now, the principal takeaway is that the hacker should be capable of deciding which microcontrollers, from among those available in the market, would satisfy the requirements imposed on each of the three microcontrollers in the pattern. Further, s/he should be intimately aware of their programming capabilities and limitations. This knowledge can then be repeatedly applied to (and grown via) multiple projects.

Figure 1 does not really reflect the usage patterns of multicore chips and/or FPGAs, nevertheless each micrcontroller shown in the figure may be considered as a core on a multicore chip or implemented as an FPGA.

Experiences with more systematic ways of selecting hardware platforms are described in [53].

2.1.3 Technological solutions

For low-level platforms, the Arduino[2] family of microcontrollers are good candidates. These come in a wide variety of configurations, suitable for varying interfacing requirements.

For medium-level platforms, some candidates are the Raspberry Pi[37] and Beaglebone[3]. These are Single Board Computers with ARM® based microcontroller and can execute the Linux operating system with or without various realtime patches.

For high-level platforms, candidates would be computers based on the Intel® Mini-ITX form factor, with multicore Atom™ processors, equipped with Solid State Drives (SSDs). Such configurations can be built with zero moving parts and are capable of executing standard desktop operating systems including Microsoft® Windows® and Linux.

Most of the freely available and open source operating systems today are not yet at the stage where they can leverage multicore chips in embedded systems for task set isolation. With Linux, for example, it is possible to set the 'CPU Affinity' of a task or process to a certain core, but that by itself in not sufficient to guarantee isolation, performance and timing characteristics. Some proprietary hypervisors are available which support isolation of tasks among cores, however their usage is very closely tied to and dependent on the specific hardware being used.

Most of the candidate platforms mentioned above have open hardware designs and can use open source software. They enjoy the benefit of extensive user communities with the associated tutorials, mailing lists and support groups, reference code and designs.

2.1.4 Suggested exercises

1. Download the source code of the Linux kernel, apply necessary patches, cross-compile and boot it on the Raspberry Pi/Beaglebone black.

2. Generate a stable PWM waveform using Linux on the Raspberry Pi/Beaglebone black as well as with an Arduino. Compare the time taken for the background reading, coding and debugging needed to perform this task. Repeat the process on both platforms, but this time for reading a quadrature encoder. Were there any differences observed? Why?

2.2 Operating systems and bootstrapping

2.2.1 Skill and solution patterns

Theoretical knowledge of the foundations and concepts of operating systems needs to be complemented by the practical ability to select a suitable OS and bootstrap it on the chosen hardware. This skill, in turn, needs to be complemented with knowledge of programming paradigms and libraries that are permitted by the OS and suitable for desired functionality. For example, given a particular microcontroller, which realtime operating system (RTOS) can be executed on it? Which standard C library functions are safe to use in a realtime environment and which ones should be avoided? Which platforms support the co-existence of realtime and non-realtime tasks and what mechanisms should be used to prevent them from interfering with each other? How should realtime tasks be prioritized on a non-preemptible kernel?

Typically open source operating system parts can be individually selected and configured to create a board specific package tailored for the needs of a particular use case. In such cases, it is an essential skill to be able to identify which packages will be needed and combine them to create the application specific OS kernel and userland utilities.

In almost every case, the source code for an embedded system needs to be cross-compiled. Setting up the cross-compilation toolchain and build environment is a valuable skill.

The usage of operating systems is recommended more often than not, over bare metal programming, wherever possible. However, the operating systems needs to be carefully matched with the hardware and it is worthwhile knowing which OSes are supported on which hardware platforms.

2.2.2 Technological solutions

The Linux kernel can be patched and configured with a variety of options to achieve varying levels of realtime behavior. These include configuration options like PREEMPT_VOLUNTARY and PREEMPT, a fully preemptive patch PREEMPT_RT[9] and dual-kernel, hard realtime mechanisms like Xenomai[46]. Each approach has a tradeoff between configuration effort, ease of programming and performance. Further, each approach imposes its own quirks and programming patterns, which must be satisfied if the desired performance is to be obtained.

Projects like Yocto[48] can be used to create custom embedded linux distributions. Alternatively, tools like Debian Multistrap[13] can be used to create partial operating systems which can then be customized for specific use cases.

For low end devices, operating systems like FreeRTOS[19] and Erika Enterprise[18] are available in the form of libraries, which can be linked with the application being developed.

2.2.3 Suggested exercises

1. Create a minimal Linux distribution based on the Xenomai kernel framework for the beaglebone/raspberry pi. Use either Yocto Project[48] or Debian Multiarch[13].

2. Create a gcc[20] based cross-compiler toolchain for generating Linux ARM® binaries on an x86 host running Microsoft® Windows®.

2.3 Execution of domain specific models

2.3.1 Skill

Experts in domains like signal processing, control engineering etc. use specific tools to model and manipulate the systems they are working on. A prime example of such a tool is Matlab®[28] and Simulink®[40], but many others exist. For better or worse, Simulink® is the *de facto* tool used by control systems engineers in the industry. Therefore, the embedded systems hacker needs the ability to take a model from such a tool and execute it in realtime when necessary. For example, a PID controller that is devised as a Simulink® model may need to be executed in realtime on a microcontroller.

It is not reasonable to expect the domain expert to manually write high-quality C or C++ code that implements his model. The onus of doing this often falls on the system integrator who has to not only transform the model to code, but also ensure that assumptions regarding the code's execution (periodicity, latency, memory etc.) are preserved. The model may need to interface with the real system's inputs/outputs and it may evolve rapidly. Therefore the model->code->execution chain needs to be as simple and painless as possible.

2.3.2 Solution patterns

There are three approaches for executing domain specific models.

1. Understand the model's construction and behavior and manually replicate it with hand-coding. This approach is fraught with risk for all but the simplest models and is rarely used in practice.

2. Automatic generation of code from the model. This requires tool support and the generated code may need to be manipulated by hand or via scripts before it can be compiled. This approach is easier, but still requires that the model assumptions regarding execution properties (periodicity, latency, memory etc.) are valid.

3. Automatic model execution, also known as rapid (control) prototyping. Certain tools provide the ability to execute the model in realtime at the "push of a button". For example, dSPACE[14] systems provide the hardware and software necessary to execute a Simulink® model in realtime. This approach largely negates the need to manually mess with generated source code and execute it under specified constraints. The technical

disadvantage with this approach is the loss of flexibility. The architecture must accommodate the hardware used for model execution and any communication with the executing model must follow the modalities dictated by the tool vendor. Quite often, these modalities are rather basic and the architecture must then include specific adaptation layers to communicate with the executing model.

Given the convenience of automatic model execution, it is used quite frequently for function development and prototyping, especially in the industry. The architecture pattern in Figure 1 supports this approach, since it defines a separate microcontroller for executing realtime code. More often than not, the realtime code is a control/signal processing model and the separate microcontroller is provided by a tool vendor e.g. dSPACE AutoBox. (Even if automatic model execution is not being used, a realtime hw+sw stack is realized in a specific way and it often makes sense to have it as a distinct part of the system architecture.)

2.3.3 Technological solutions

Vendors of tools like Simulink® provide fairly capable code generators like Embedded Coder®[27] for transforming models to source code. The generated code may then be integrated into the rest of the application code, either manually or via the build system. One of the open source equivalents of Simulink®, SciCos[39], has a variety of code generators but in general, they are either immature or with limited functionality.

For automatic model execution, the industrial heavyweight is dSPACE systems[14] which offer tightly integrated hardware and software solutions. The MathWorks also has a solution named Simulink Real-Time™[41] which offers some more flexibility in selecting the hardware. Both these solutions utilize proprietary operating systems which lack the standard C/C++ language runtimes. This imposes a strong limitation on the Simulink® model: it may not incorporate any custom C/C++ code that depends on the availability of the standard language runtime. In practice, most code that does anything useful depends on the language runtime.

A lightweight solution for automatic model execution is to utilize an Arduino[2] board. There exist blocksets and code generators for both Matlab®/Simulink® and Scicos[39] that essentially offer a "push button interface" to execute models on the Arduino, converting the Arduino into a "poor man's dSPACE".

2.3.4 Suggested exercises

1. Design a DC motor speed controller in Simulink®. Manually convert it to C++ code and execute on an Arduino. Then, use Simulink's Arduino support to automatically execute the model on the arduino. Compare experiences with respect to time needed and access to debugging information.

2. Communicate information via ethernet to/from a model executing on a Simulink® Real-Time target™. The information could be in the form of C/C++ data structures containing floating point numbers and text strings. Try to use higher level communication libraries that automatically perform data de/serialization and transfer on top of UDP/TCP.

2.4 Communication

2.4.1 Skill

An architecture involving multiple microcontrollers or computer systems will, sooner than later, require information to be communicated among them. Communication methods and protocols are almost as varied as the hw+sw stacks themselves and decisions need to be made regarding which ones are appropriate in a given situation.

In matters of communication, a useful principle to follow is: For any given set of constraints, the communication method selected should require minimum programmer effort for communicating the desired data. Thus, although a programmer can manually pack and unpack byte sequences into communication buffers, it is desirable to use mechanisms and libraries that operate at the level of data structures or class objects. In the latter case, the sender would simply pass a class object to the communication library, which would then be reproduced as an equivalent object in the receiver. The programmer would not care about how the object de/serialization takes place, which wire transfer protocol is used, etc. As objects grow more complex (for example, if they contain nested pointers to other objects), having a communication library handle all the intricacies becomes more and more appealing. Unfortunately, it is a common observation that students are barely introduced to UDP/TCP client-server programming and rarely are the higher level communication mechanisms taught.

When programming a communications subsystem, distinction is usually made between the *packaging* and *transfer* of data. Packaging specifies the format in which the data is communicated. It may include the wire-level representation of the data. Transfer, on the other hand refers to how the packaged data is communicated across a network. For example, a data structure may be packaged as a BSON[7] object and transferred from a client to a UDP server. It is a valuable skill to know which packaging and transfer mechanisms should be used in a given situation.

2.4.2 Solution patterns

Solutions for data packaging can be broadly classified based on whether the resulting data package is in plain text or binary formats. Text based packaging typically utilizes structured, human readable information and is preferred wherever the additional overhead of converting binary data to ascii representation is not prohibitive. Binary packaging is utilized where efficiency of communication has a higher priority than human readability. This is frequently the case in embedded systems where resources are at a premium. The obvious disadvantage of binary packaging is that making sense of captured data packets during debugging phases is not straightforward. It is not uncommon to utilize a text based package format during the debugging step and transition to binary at a later point. In fact, data description languages like ASN.1 permit *encoding/decoding* the same data structure into a variety of text based and binary formats.

Patterns for data transfer can be categorized by topology and scalability characteristics. These include client-server, push-pull, publish-subscribe, router-dealer and so on. Each pattern dictates how data flows from the source(s) to consumer(s).

Taking a long step back from the high level communication concepts discussed so far, there are times when simple

bit banging suffices (or is the only option) as the communication method.

2.4.3 Technological solutions

There exist a plethora of solutions for data packaging. These take the form of open (or closed) specifications and library implementations that perform the actual (un)packaging of data. Commonly used text based examples are XML, JSON[16], and ASN.1[1] with XER[47] encoding. Common binary packaging solutions include BSON[7], XDR[45] and Google protocol buffers[24].

Libraries like Boost serialization[6] can be used to make class objects serialize-able to a variety of archive formats.

In addition to classical UDP/TCP client-server programming, newer communication libraries and frameworks like ZeroMQ[50] and various DDS[44] implementations provide less painful ways to realize many different transfer methods. In fact, based on anecdotes, one may go so far as to say that once a hacker gains experience with a transport layer like ZeroMQ, it is difficult to go back to traditional socket programming.

Solutions like CORBA[11], ZeroC ICE[49] and the aforementioned DDS perform both data transfer and packaging. However, their usage is limited because they are "heavy" solutions requiring substantial computing resources.

2.4.4 Suggested exercises

1. Write code that converts a C data structure to a BSON object. Next, transfer this object using a publish-subscribe mechanism to at least two different computers, where the object's contents should be read out.
2. Put the publisher part of the above code in a Simulink® S-function block. Include the S-function block in a Simulink® model and publish values of different model variables to the two subscribers.

2.5 Data logging and visualization

2.5.1 Skill

Logging data and visualizing that data, either in realtime or as a replay, is a crucial aspect in the development of embedded systems. This can be required in the early debugging stage, the calibration and evaluation stage, as well as in the post-deployment stage. Therefore, knowing the various mechanisms of logging information and having the ability to recall, manipulate and visualize the information in interesting ways is an essential skill for the embedded systems hacker.

By its nature, data logging involves input/output operations. These operations are usually asynchronous and non-deterministic on most systems and involve a lot of "under the hood" activity like buffering, disk defragmentation, flushing of memory buffers, disk controller seek times etc. As a result, unless special realtime capable techniques are used, datalogging is a non-realtime activity. This fact immediately puts some requirements on the system architecture, the least of which is the separation of realtime and non-realtime tasks and having some form of communication between them.

2.5.2 Solution patterns

Given the non-realtime nature of datalogging (and even when realtime dataloggers are present), it is a common architectural pattern to have a separate thread/task to perform

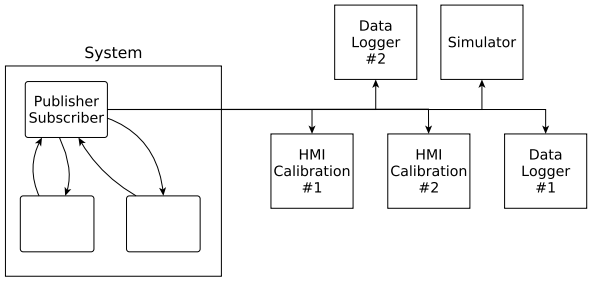

Figure 2: A publisher based logging pattern

the logging activity. The thread may write the data to disk, display it in a convenient way, or stream it out over the network. It is often better if logging happens in its own process (so as to not disturb the realtime code timing) and even better if the process is on an entirely different computer.

In all cases, it is of extreme convenience if the logging code can treat the data it receives as a "black box" i.e. if it need not know the internal structure of the data being received. The justification for this is two-fold. Firstly it is about limiting the effect of changes. It is a nuisance to change the logging code every time someone changes the contents of the data structure being logged. Secondly, it is about scale. If a hundred data structures are being logged, the logger thread needs to know all their details to write them properly to disk or to a network socket. However, if the data object being logged is self-describing or has de/serialize methods, then the datalogger can essentially call `object.serialize()` for whichever object it receives (even better if all received objects inherit from a common parent having `serialize()` as an abstract method) and place the result into a `send()` call and be done with it. At the receiving end, the reverse process occurs. The logging code needs zero knowledge of the object it is logging, apart from the fact that it can be `serialize()`d and `deserialize()`d.

For most embedded systems, the data to be logged can be categorized as time-triggered and event-triggered. Time-triggered data is usually generated periodically and its contents and size are known in advance. Examples of this are sensor readings, actuator inputs etc. Event-triggered data typically arises in a non-deterministic way, depending on the event which triggers it. Exceptions and error handling code can often generate such data. The qualitative difference between event-triggered data and time-triggered data is that in the former case, the contents and their size may not be known in advance. For example, if the code triggers a segmentation fault and this is caught by an exception handler, the resulting error message string may be accompanied by the unwound stack frames at that point. The size of this information is not always predictable.

Given the existence of publish-subscribe transfer patterns, a common data logging pattern is as shown in Figure 2. Basically, all the data to be logged in a system is collected by a publisher task and streamed out. Downstream, there may be any number of subscribers on any number of computers that may receive, filter and operate on the received data.

Yet another pattern is shown in Figure 3. This pattern basically "abuses" the metaphor of a standalone database and client libraries to access it. Essentially, the logging task links against a client library of an object database and dumps the

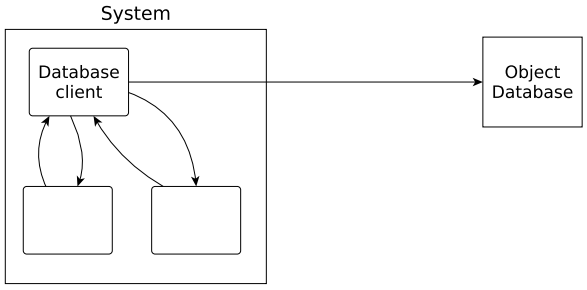

Figure 3: A database based logging pattern

received objects into the database, which may or may not be on another computer.

A third pattern is to utilize an "in-memory" database in the embedded application and utilize the database's replication facilities to synchronously replicate the database on a different computer. If the performance and application timing characteristics permit this scheme to work, datalogging essentially comes "for free".

2.5.3 Technological solutions

As an application grows, it usually requires a proper logging framework supporting log levels, various logging backends, thread safety and more. Getting this right is harder than it first seems, so it is recommended to use a generic logging framework like Boost.log[5].

The format of the logged data depends on storage size requirements, efficiency and quite often on the tool that will be used to examine and work with the logged information. Traditionally, timeseries data may be logged in the comma separated value a.k.a csv format, but options like netCDF[31], HDF5[25] exist as well, which are especially suited for storing and selectively accessing high volume data.

Tools like Matlab® and qtiplot[36] are useful to analyze, plot and replay logged data. Depending on task and programmer skills, one of the most useful tools to work with logged data is simply the Python programming language, together with the numpy[32] package for scientific computing and matplotlib[29] for plotting. In the open source community, the tools GNU Octave and gnuplot[23] are very popular. These tools allow for the creation of custom scripts that can manipulate logged data in desired ways.

Logging to databases is accomplished more easily with the so called 'nosql' databases, These databases directly store 'documents' which encapsulate and encode structured data, or key/value pairs. In contrast with sql databases, the overhead of converting/querying an object's contents to a relational database schema are avoided. MongoDB[30] is a popular document database, while BerkeleyDB[4] works with key-value pairs. If sql is needed, sqlite[42] provides a lightweight, self-contained, embeddable, transactional database. Both sqlite and berkeleydb can be used as "in-memory" databases, if that is what the application demands.

Vendors of commercial tools may offer their own logging infrastructure tuned to work with their tools. For example the OpenSplice DDS[33] implementation provides a connector which captures desired data from the publish/subscribe bus and saves it to an enterprise database.

2.5.4 Suggested exercises

1. Revisit exercise 2 from section 2.4.4. Write subscribers that dump the received Simulink® values of model variables to mongoDB and live-plot specific variables as a ticker plot across the screen.

2. Write a Python program to query the database from the above exercise and plot user defined variables to the screen.

2.6 Build systems

2.6.1 Skill

The build system has the task of compiling the software sources into executable binaries for the target platform. As the application becomes more complex, so does the build system. It is not uncommon to find that the build system needs to generate intermediate binaries along the way, which pre-process part of the source code, or call scripts that autogenerate some code prior to building the whole set of sources. Some build systems need to utilize different compilation and optimization options depending on whether a debug or release deployment is made. Yet others may have the requirement of compiling the same set of sources for different hardware platforms and operating systems. Looking at the build systems of complex software like the Linux kernel or some middleware like OROCOS[34] corresponds to an education in itself.

The embedded system hacker needs to go beyond the simple "Projects & Workspaces" generated by Integrated Development Environments (IDEs) like Eclipse[15] and gain the capability to create build systems that can take care of complex software dependencies, order of compilation, autodetection of installed libraries and headers and using their correct versions, prompting the user for missing supporting software and using platform specific compilation toolchains.

2.6.2 Solution patterns

The simplest build system is that autogenerated by the IDE in use. Most mature IDEs autodetect the compiler installed on the system, probe standard filesystem locations for installed libraries and compile the files added to the "Project". Such simple systems quickly start showing their limitations, in addition to the more serious offense of breeding generations of student programmers who have no clue how the build system works behind the scenes.

Where IDEs are not used (and sometimes even where they are), the most common pattern is to define a special file that contains a series of build instructions. These instructions define the source files and rules to generate the final executable and intermediate object files. It is possible to define compiler and linker flags, additional arguments and dependencies between the source files. This file is then processed by a special tool which then builds the whole application. The venerable UNIX Makefile is the most prominent example of this category.

More flexible build patterns go beyond Makefiles. This approach contains macros and scripts to automatically configure software source packages. The build system can easily adapt to a variety of software already available on the host system. It can package the executables into deployment-ready formats (installers etc.) as well as run unit-tests and

make other basic checks. Along the way, Makefiles are usually autogenerated as one step in building the software.

Very large and complex applications may require custom wrapper scripts around all the previously mentioned patterns, which may include additional and more user-friendly functionality like downloading missing software dependencies.

Finally, a recent trend in building software is Continuous Integration[10], which utilizes build servers, to automatically run unit tests periodically or even after every commit and report the results to the developers via email, automatically generated web pages and even in the form of Internet Relay Chat "bots".

2.6.3 Technological solutions

Almost all development approaches involve using the 'make' utility in some form or the other, somewhere in the build process. On Linux GNU make[22] is the default utility, but specific libraries may have their own make tools with additional capabilities. An example is qmake[35] that is the preferred tool when building software utilizing the Qt libraries.

Somewhat similar to qmake, but with greater capabilities, is a suite of utilities collectively called the GNU build system[21], whose most important components are Autoconf, Automake and Libtool. Although very capable, the GNU build system can be rather complex and difficult to use.

CMake[8] is a more recent build tool which is a cross-platform and works on Linux, Windows, Mac OSX and many other UNIXes. Among other things it offers a variety of "frontends" for better managing the build process and some of these frontends are also graphical.

One of the most popular Continuous Integration tools is the open source Jenkins[26], which is used by several large open source projects and commercial companies.

2.6.4 Suggested exercises

1. Use CMake to create a build system for your project.

2. Explore how the build system for the open source middleware, OROCOS[34], works.

3. DISCUSSION AND CONCLUSIONS

It must be emphasized that many of the skills mentioned here are indeed introduced in various courses in embedded systems education. However, their theoretical depth is not always balanced with practical depth. One reason for this could be that laboratory exercises have traditionally aimed at reinforcing the theoretical concepts, rather than providing the breadth and depth needed to build systems in practice. Thus, a lab session is less likely to ask students to bootstrap a platform, and more likely to provide a carefully prepared platform, environment and skeleton code using which the students can see how the theory is reflected in practice.

Two main findings from both [17] and the WESE workshop where this paper was initially presented, were that 1: There is a shortage of education platforms and 2: There is a need for an increased university and industrial collaboration. Lab platforms and ecosystems in universities often lag behind the state of the art, because universities lack resources to develop and and maintain top notch platforms, nor do they collaborate with other universities to reduce and distribute the required effort. However, we believe that increasing cooperation between universities and the industry, holds a potential to simultaneously address the problem of platform familiarity, especially when industrial grade equipment and software is used during the collaboration.

Corresponding challenges for educators are to keep track of evolving technology, find a suitable balance in incorporating new technology in the syllabi, increase practical depth and make the graduating students useful to industry "straight off the bat". The last two challenges have been tackled with good success by the CDIO approach[43] and by problem-based and project-organized courses in embedded systems[51]. Such courses, when conducted in cooperation with the industry, are showing good promise[54] in achieving the goal of equipping students with practical skills that are immediately useful for both industry and research. Indeed, the final courses in mechatronics education at KTH in Stockholm are now almost exclusively project courses, with the projects provided by the Swedish industry. In the context of such a project course, a student acquires the skills mostly by self-study. In an attempt to inculcate some of the skills in a more traditional format, a precursor course in embedded systems at KTH (MF2044) was modified to include execution of domain specific models in one of the lab sessions. The students were tasked to control the speed of a DC motor, first by manual coding based on a Simulink model, then by direct execution of a Simulink model on an Arduino platform. A comparison of the two approaches enabled the students to appreciate the flexibility of the former, with the ease of the latter, and methods of combining the two approaches using, for example, Simulink's S-functions. It was later observed that a majority of the students exhibited good judgment in applying these skills during the subsequent project course.

In summary, we have preliminary evidence to believe that a combination of project and traditional courses can inculcate the described skills in the context of university education. Given the spread of the presented skill set, and the depth and variation in technologies involved, it remains a challenge to find an optimal pattern for imparting such education. In balancing between theory and practice, educators need to stimulate potential hackers to become technical leaders and architects. The skills presented in this paper provide guidance for course developers. Pursuing such efforts require exploiting best educational practices and establishing forms for closer industry collaboration.

References

[1] Abstract Syntax Notation One ASN.1. `http://www.etsi.org/technologies-clusters/technologies/protocol-specification/asn-1`.

[2] Arduino - Open Source Electronic Platform. `http://arduino.cc`.

[3] BeagleBone Black. `http://beagleboard.org/black`.

[4] BerkeleyDB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`.

[5] Boost Log v2. `http://www.boost.org/doc/libs/1\_55\_0b1/libs/log/doc/html/index.html`.

[6] Boost Serialization library. http://www.boost.org/doc/libs/1_55_0/libs/serialization/doc/index.html.

[7] BSON - Binary JSON. http://bsonspec.org.

[8] CMake - Cross Platform Make. http://cmake.org.

[9] CONFIG_PREEMPT_RT patch to make Linux kernel fully preemptible. https://rt.wiki.kernel.org.

[10] Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html.

[11] CORBA - Common Object Request Broker Architecture. http://www.corba.org.

[12] CyPhERS project deliverable D5.2: CPS: Significance, Challenges and Opportunities. http://www.cyphers.eu/sites/default/files/D5.2.pdf.

[13] Debian Multistrap - Bootstrap a Debian system. https://wiki.debian.org/Multistrap.

[14] dSPACE Systems. https://www.dspace.com.

[15] Eclipse IDE. http://eclipse.org.

[16] ECMA-404 the JSON Data Interchange Standard. http://json.org.

[17] Engineering education in embedded systems. ICES workshop summary, August 26, 2014. http://www.ices.kth.se/events.aspx?pid=3&evtKeyId=295145e8aefa46dfa0277baeb820e439.

[18] ERIKA Enterprise - Certified OSEK/VDX Open Source RTOS. http://erika.tuxfamily.org/drupal.

[19] FreeRTOS. http://www.freertos.org.

[20] GCC, the GNU Compiler Collection. http://gcc.gnu.org.

[21] The GNU Build system. http://www.gnu.org/software/automake.

[22] GNU Make. http://www.gnu.org/software/make.

[23] Gnuplot. http://www.gnuplot.info.

[24] Google Protocol Buffers. https://code.google.com/p/protobuf.

[25] HDF5 Data Format. http://www.hdfgroup.org/HDF5.

[26] Jenkins - an extendable open source continuous integration server. http://jenkins-ci.org.

[27] MathWorks Embedded Coder. http://www.mathworks.se/products/embedded-coder.

[28] MATLAB. http://www.mathworks.se/products/matlab.

[29] matplotlib: Python plotting. http://matplotlib.org.

[30] mongoDB. http://www.mongodb.info.

[31] Network Common Data Form (NetCDF). http://www.unidata.ucar.edu/software/netcdf.

[32] NumPy scientific computing with Python. http://www.numpy.org.

[33] Opensplice DDS. http://www.prismtech.com/opensplice.

[34] OROCOS - open robot control software. http://orocos.org.

[35] qmake manual. http://qt-project.org/doc/qt-5/qmake-manual.html.

[36] QtiPlot - Data Analysis and Scientific Visualisation. http://www.qtiplot.com.

[37] Raspberry Pi. http://www.raspberrypi.org.

[38] RFC1392 - Internet Users' Glossary. http://tools.ietf.org/html/rfc1392.

[39] Scicos: Block diagram modeler/simulator. http://scicos.org.

[40] Simulink. http://www.mathworks.se/products/simulink.

[41] Simulink Real-Time. http://www.mathworks.se/products/simulink-real-time.

[42] SQLite. http://sqlite.org.

[43] The CDIO initiative. http://www.cdio.org.

[44] The OMG Data Distribution Service for Real-Time Systems (DDS). http://portals.omg.org/dds.

[45] XDR: External Data Representation Standard. http://tools.ietf.org/html/rfc4506.

[46] Xenomai: Real-Time Framework for Linux. https://xenomai.org.

[47] XML encoding rules (XER) for ASN.1. http://www.itu.int/en/ITU-T/asn1/Pages/xer.aspx.

[48] Yocto Project: Open Source embedded Linux build system. https://www.yoctoproject.org.

[49] ZeroC Internet Communications Engine. http://www.zeroc.com.

[50] ZeroMQ. http://zeromq.org.

[51] M. Grimheden and M. Törngren. What is embedded systems and how should it be taught?—results from a didactic analysis. *ACM Trans. Embed. Comput. Syst.*, 4(3):633–651, Aug. 2005.

[52] J. Mårtensson, A. Alam, and S. Behere. The Development of a Cooperative Heavy-Duty Vehicle for the GCDC 2011: Team Scoop. *IEEE Transactions on Intelligent Transportation Systems*, 13(3):1033–1049, Sept. 2012.

[53] F. Salewski and S. Kowalewski. Hardware platform design decisions in embedded systems: A systematic teaching approach. *SIGBED Rev.*, 4(1):27–35, Jan. 2007.

[54] M. Törngren, M. Grimheden, and N. Adamsson. Experiences from large embedded systems development projects in education, involving industry and research. *SIGBED Rev.*, 4(1):55–63, Jan. 2007.