Using Simulation Tools for Embedded Software Development

Class #410, Embedded Systems Conference, Silicon Valley 2008

Jakob Engblom
Virtutech
Drottningholmsvägen 14
SE-112 42 Stockholm
Sweden
e-mail: jakob.engblom@virtutech.com

Simulation is an engineering method used when it is inconvenient, expensive, impractical, or plain impossible to try things in reality. It reduces the cost of experiments while increasing insight and control. It makes systems that do not yet exist accessible. It cuts lead times and improves product quality.

This class gives an overview of how simulation (and virtualization) can be used to help embedded software development, covering simulation of processors, boards, networks, systems, user interfaces, and the environment. In particular, we will deal with how to build a concrete simulation system for an embedded board that can run the real software.

Introduction

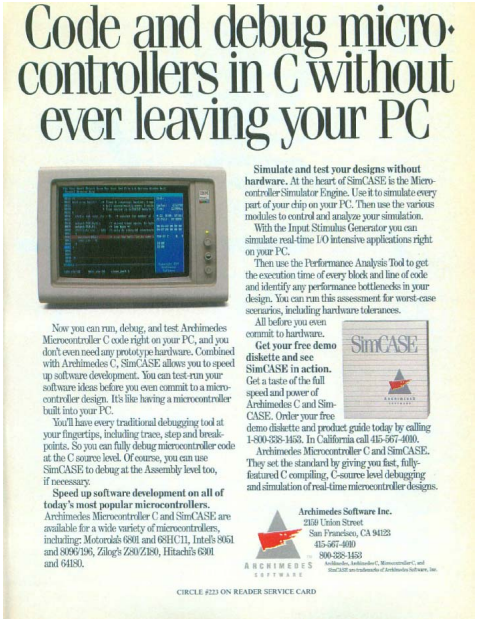
Simulation as a tool has been used for a long time in many areas of science and technology. The use of weather system simulation to predict the weather and mechanical system simulation to predict the behavior of aircraft in flight are assumed as given today. In car design, the use of virtual crash testing improves the safety of cars while radically reducing the number of physical prototypes that need to be built and tested.

Simulation is used whenever trying things in the physical world would be inconvenient, expensive, impractical or plain impossible. Simulation allows experimenters to try things with more control over parameters and better insight into the results. It reduces the cost of experiments, and makes it possible to work with systems that do not yet exist in physical form. It cuts lead times and improves product quality. In a sense, we use simulation and virtual systems because reality sucks. Ironically, while simulation is almost universally implemented as software on computers, the use of simulation to develop computer software itself is still quite rare.

Here, we will look specifically how custom embedded computer systems can be simulated for the benefit of embedded software development. We will not address the use of simulation to do hardware-software codesign, system-on-chip design, and other hardware-oriented tasks.

Note that as embedded systems start using multiple processors and multicore processors, using simulation and "virtual prototypes" for software development is one of the best ways to get a grip on the complex debugging and diagnostics issues arising from the concurrent execution of multiple threads of control on multiple processors.

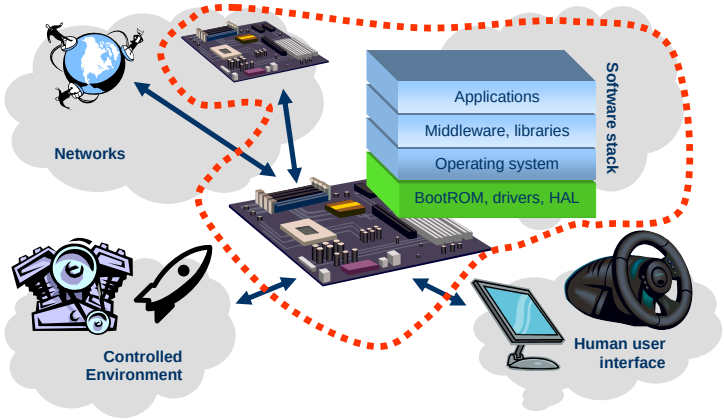
Finally, we should acknowledge that the idea of using simulated computers for software development is not new. Looking at the first issue of Embedded Systems Programming from 1988 they have both advertisements and a product review of a simulator for embedded systems. The arguments put forward are the same as those of today. Thanks to Jack Ganssle, www.ganssle.com, I can show you a picture of an ad from 1988.



Simulating a Computer System

An embedded computer system can be broken down into five main parts.

- The computer board itself: the piece of hardware containing one or more processors, executing the embedded software.
- The software running on this computer board. This includes not just the user applications, but also the boot ROM or BIOS, hardware drivers, operating system, and various libraries and middleware frameworks used to implement the software functions.
- The communications network or networks that the board is connected to, and over which the software communicates with software on other computers.
- The environment in which the computer operates and that it measures using sensors and affects using actuators.
- The user interface exposed to a human user of the system.



For the rest of this article, we will refer to the system being simulated as the **target** system (common with traditional cross-compilation nomenclature), and the development PC or workstation as the **host**. Obviously, not all target systems feature all of the parts listed above, but most feature most of them. A simulation effort for an embedded system can focus on just one of the parts. It is quite common to mix simulated and physical parts, to achieve "partial reality."

We will go deeply into the issues of simulating a board to run the real software, and how to quickly create the hardware models needed to run the unmodified software.

Abstraction vs. Detail

A key insight in building simulations is that you must always make a trade-off between simulator detail and the scope of the simulated system. Looking at some extreme cases, you cannot use the same level of abstraction when simulating the evolution of the universe on a grand scale as when simulating protein folding. You can always trade execution time for increased detail or scope, but assuming you want a result in a reasonable time scale, compromises are necessary.

A corollary to the abstraction rule is that simulation is a workload that can always use maximum computer performance (unless it is limited by the speed of interaction from the world or users). A faster computer or less detailed model lets you scale up the size of the system simulated or reduce simulation run times. In general, if the processor in your computer is not loaded to 100%, you are not making optimal use of simulation.

The high demands for computer power used to be a limiting factor for the use of simulation, requiring large, expensive, and rare supercomputers to be used. Today, however, even the cheapest PC has sufficient computation power to perform relevant simulations in reasonable time. Thus, the availability of computer equipment is not a problem anymore, and simulation should be a tool considered for deployment to every engineer in a development project.

Simulating the Environment

Simulation of the physical environment is often done for its own sake, without regard for the eventual use of the simulation model by embedded software developers. It is standard practice in mechanical

and electrical engineering to design with computer aided tools and simulation. For example, control engineers developing control algorithms for physical systems such as engines or processing plants often build models of the controlled system in tools such as MatLab/Simulink and Labview. These models are then combined with a model of the controller under development, and control properties like stability and performance evaluated. From a software perspective, this is simulating the specification of the embedded software along with the controlled environment.

For a space probe, the environment simulation could comprise a model of the planets, the sun, and the probe itself. This model can be used to evaluate proposed trajectories, since it is possible to work through missions of years in length in a very short time. In conjunction with embedded computer simulations, such a simulator would provide data on the attitude and distance to the sun, the amount of power being generated from solar panels, and the positions of stars seen by the navigation sensors.

When the mechanical component of an embedded system is potentially dangerous or impractical to work with, you absolutely want to simulate the effects of the software before committing to physical hardware. For example, control software for heavy machinery or military vehicles are best tested in simulation. Also, the number of physical prototypes available is fairly limited in such circumstances, and not something every developer will have at their desk. Such models can be created using modeling tools, or written in C or C++ (which is quite popular in practice).

In many cases, environment simulations can be simple data sequences captured from a real sensor or simply guessed by a developer.

It should be noted that a simulated environment can be used for two different purposes. One is to provide “typical” data to the computer system simulation, trying to mimic the behavior of the final physical system under normal operating conditions. The other is to provide “extreme” data, corresponding to boundary cases in the system behavior, and “faulty” data corresponding to broken sensors or similar cases outside normal operating conditions. The ability to inject extreme and faulty cases is a key benefit from simulation.

Simulating the Human User Interface

The human interface portion of an embedded device is often also simulated during its development. For testing user interface ideas, rapid prototyping and simulation is very worthwhile and can be done in many different ways. One creative example is how the creator of the original Palm Pilot used a wooden block to simulate the effect of carrying the device. Instead of building complete implementations of the interface of a TV, mobile phone, or plant control computer, mockups are built in specialized user interface (UI) tools, in Visual Studio GUI builder on a PC, or even PowerPoint or Flash. Sometimes such simulations have complex behaviors implemented in various scripts or even simple prototype software stacks. Only when the UI design is stable do you commit to implementing it in real code for your real device, since this typically implies a greater programming effort.

In later phases of development, when the hardware user interface and most of the software user interface is done, a computer simulation of a device needs to provide input and output facilities to make it possible to test software for the device without hardware. This kind of simulation runs the gamut from simple text consoles showing the output from a serial port to graphical simulations of user interface panels where the user can click on switches, turn knobs, and watch feedback on graphical dials and screens. A typical example is Nokia’s Series 60 development kit, which provides a virtual mobile phone with a keypad and small display. Another example is how virtual PC tools like VmWare and Parallels map the display, keyboard, and mouse of a PC to a target system.

In consumer electronics, PC peripherals are often used to provide live test data approximating that of a real system. For example, a webcam is a good test data generator for a simulated mobile phone containing a camera. Even if the optics and sensors are different, it still provides something better than static predetermined images. Same for sound capture and playback – you want to hear the sound the machine is making, not just watch the waveform on a display.

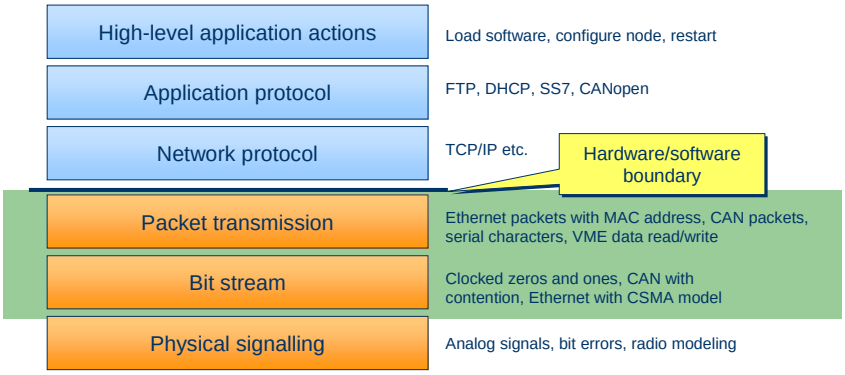
Simulating the Network

Most embedded computers today are connected to one or more networks. These networks can be internal to a system; for example, in a rack-based system, VME, PCI, PCI Express, RapidIO, Ethernet, I^2C, serial lines, and ATM can be used to connect the boards. In cars, CAN, LIN, FlexRay, and MOST buses connect body electronics, telematics, and control systems. Aircraft control systems communicate over special bus systems like MIL-STD-1553, ARINC 429, and AFDX.

Between the external interfaces of systems, Ethernet running internet standards like UDP and TCP is common. Mobile phones connect to headsets and PCs over Bluetooth, USB, and IR, and to cellular networks using UMTS, CDMA2000, GSM, and other standards. Telephone systems have traffic flowing using many different protocols and physical standards like SS7, SONET, SDH, and ATM. Smartcards connect to card readers using contacts or contact-less interfaces. Sensor nodes communicate over standard wireless networks or lower-power, lower-speed interfaces like Zigbee.

Thus, existing in an internal or external network is a reality for most embedded systems. Due to the large scale of a typical network, the network part is almost universally simulated to some extent. You simply cannot test a phone switch or router inside its real deployment network, so you have to provide some kind of simulation for the external world. You don't want to test mobile phone viruses in the live network for very practical reasons. Often, many other nodes on the network are being developed at the same time. Or you might just want to combine point simulations of several networked systems into a single simulated network.

Network simulation can be applied at many levels of the networking stack. The picture below shows the most common levels at which network simulation is being performed. The two levels highlighted in green are the ones that are most useful for embedded software work on a concrete target model.



The most detailed modeling level is the **physical signal** level. Here, the analog properties of the transmission medium and how signals pass through it is modeled. This makes it possible to simulate radio propagation, echoes, and signal degradation, or the electronic interference caused by signals on a CAN bus. It is quite rarely used in the setting of developing embedded systems software, since it complex and provides more details than strictly needed.

Bit stream simulation looks at the ones and zeroes transmitted on a bus or other medium. It is possible to detect events like transmission collisions on Ethernet and the resulting back-off, priorities being clocked onto a CAN bus, and signal garbling due to simultaneous transmissions in radio networks. An open example of such a simulator is the VMNet simulator for sensor networks. Considering the abstraction levels for computer system simulation discussed below, this is at an abstraction level similar to cycle-accurate simulation. Another example is the simulation of the precise clock-by-clock communication between units inside a system-on-a-chip.

Packet transmission passes entire packets around, where the definition of a packet depends on the network type. In Ethernet, packets can be up to 65kB large, while serial lines usually transmit single bytes in each "packet". It is the network simulation equivalent of transaction-level modeling, as discussed below for computer boards. The network simulation has no knowledge of the meaning of the packets. It just passes opaque blobs of bits around. The software on the simulated system interacts with some kind of virtual network interface, programming it just like a real network device. This level is quite scalable in terms of simulation size, and is also an appropriate level at which to connect real and simulated networks. Common PC virtualization software like VMware operates at this level, as do embedded-systems virtualization tools from Virtutech, Synopsys, and VaST.

Ignoring the actual structure of packets on the network, networks are often simulated at the level of **network protocols** like TCP/IP. The simulated nodes use some socket-style API to send traffic into a simulated network rather than a real network. Such a simulation becomes independent of the actual medium used, and can scale to very large networks. The network research tool NS2 operates at this level, for example. It is also a natural network model when using API-level simulation of the software, as discussed below.

Application protocol simulation simulates the behavior of network services and other nodes. Tools simulate both of the network protocols used and the application protocols built on top of them. Such simulation tools embody significant knowledge of the function of real-world network nodes or network

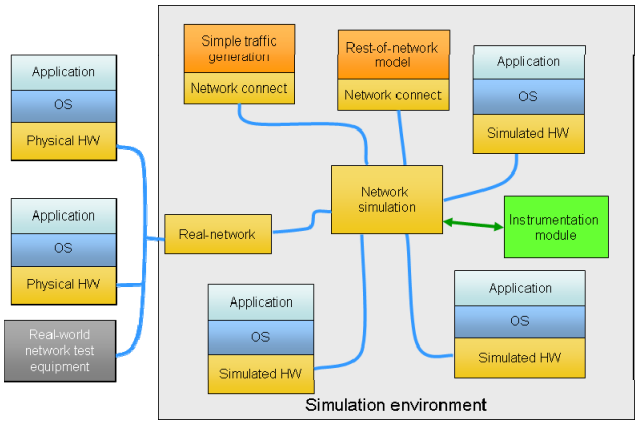
subsystems. They offer the ability to test individual network nodes in an intelligent interactive environment, a concept often known as **rest-of-network** simulation. Vector Informatik's CAN tools are a typical example of such tools.

Finally, some high-level simulations of networked systems work at the level of **application actions**. In this context, we do not care about how network traffic is delivered, just about the activities they result in. It is a common mode when designing systems at the highest level, for example in UML models.

The level of abstraction to choose depends on your requirements, and it is often the case that several types of simulators are combined in a single simulation setup. The picture below shows a complex setup that forms a superset of most real-world deployments.

- Some simulated nodes running the real software for the embedded system.
- A rest-of-network simulator providing the illusion of many more nodes on the network.
- A simple traffic generator that just injects packets according to some kind of randomized model.
- An instrumentation module that peeks on traffic without being visible on the network, showing the advantage of simulation in inspection.
- A connection to the real-world network on which some real systems are found, communicating with the simulated systems.

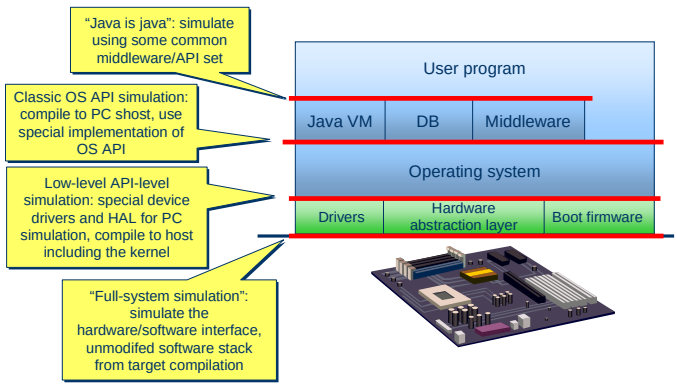
Real-world network test machines are the types of specialized equipment used today to provide testing of physical network nodes. Thanks to a real-network bridge, they can also be used with simulated systems.



Simulating the Computer System and its Software

Now we get to the core of the system: the board containing one or more processors, and the software running on that board. Since the computer board hardware and its software are so closely related, we consider the simulation of them together. The central goal is to use a PC or workstation to develop and test software for the target embedded system, without resorting to using physical hardware.

The picture below illustrates the most common levels of simulation from a software perspective:



It sometimes makes sense to program and test embedded software **against an API also found on the workstation**. Java programs and programs using some specific database API or a standard middleware like CORBA can sometimes be tested in this manner. The assumption is that the behavior of these

high-level APIs will be sufficiently similar on the host and target systems. This is not necessarily true; for example, the behavior of a Java machine on a mobile phone is not likely to be the same as on a desktop, due to memory and processing power constraints, and different input and output devices.

It is also possible to simulate some aspects of user programs with no model of the target system at all. For example, control models as discussed above are interesting to study regardless of the precise details of the target system software stack. Using UML and state charts to model pieces of software makes it possible to simulate the software on its own, without considering much of the target at all. Minimal assumptions are made about the properties of the underlying real-time operating system and target in these models.

A common and popular level of simulation is **API-level simulation** of a target operating system. This is also known as **host-compiled simulation**, since the embedded code is compiled to run on the host and not on the target system. This type of simulation is popular since it is conceptually simple and easy to understand, and is something that an embedded developer can basically create on his or her own in incremental steps. It also allows developers to use popular programming and debug tools available in a desktop environment, like Visual Studio, Purify and Valgrind.

It also has several drawbacks. First, it often ends up being quite expensive to maintain the API-level simulation and separate compilation setup required. Second, the behavior of the simulated system will differ in subtle but important details from the real thing. Details like the precise scheduling of processes, size of memory, behavior of memory protection, availability of global variables, and target compiler peculiarities can make code that runs just fine in simulation break on the real target. Third, it is impossible to make use of code only available as a target binary. Fourth, complex actions involving the hardware and low-level software, such as rebooting the system, are very hard to represent.

Most embedded operating-system vendors offer some form of host-compiled simulation. For example, WindRiver's VxSim and Enea's OSE Soft Kernel fit in this category. There are also many in-house implementations of API-level simulations, both for in-house and externally sourced operating systems. The experience with such tools range from the very successful, where minimal debugging needs to be done on the target after the switch-over to the real target, to utter failures where some approximation in the API-level simulation did not work, and the code had to be extensively debugged on the physical target.

Trying to resolve some of the issues with API-level simulation, sometimes the hardware-independent part of the embedded operating system is used, together with a simulation of the hardware-dependent parts. This is basically **paravirtualization**, where the device drivers and hardware-abstraction level of an operating system is replaced with simplified code interacting with the host operating system rather than actual hardware. A paravirtual solution provides better insight into the behavior of the operating system kernel and services, but is still compiled to run on the host PC. It also requires access to the operating-system source code, and a programming effort corresponding to creating a new board-support package (BSP).

Standalone instruction-set simulators (ISS) are common and established in embedded compiler and debug toolsets. Such simulators usually simulate only the user-level instruction set of a processor, and lets simple programs that do not really do I/O be run on the host. Modeling of peripheral devices is typically limited to providing sequences of bytes to be read from certain memory locations. Operating systems can not be run on such simulators, since more complex and necessary devices like timers and serial ports are missing. The simulators are also typically quite slow, since there is little value gained from making them faster. IDEs from vendors like IAR and GreenHills are typical examples of development tools including a basic ISS.

Full-System Simulation

Finally, the problem can be attacked at the **hardware/software** interface level. Here, a **virtual target system** is created which runs the same binary stack (from boot firmware to device drivers and operating system) as the physical target system. This is achieved by simulating the instruction set of the actual target processor, along with the programming interface of the peripheral devices in the system and their behavior. The technical term for this is **full-system simulation**, since you are indeed simulating the full system and not just the processor. It is also the main technology that we will focus on in this paper.

The crucial and defining property of this type of simulation is that all software is the same as on the physical system, using the same device drivers and BSP code. This means that the type of processor, memory sizes, processor speeds, device memory map, and other aspects have to precisely match the target.

It is common practice to develop the device drivers and BSP ports using virtual target systems in cases where the physical hardware is not yet available. We also avoid the need to maintain a separate build setup and build chain for simulation, since it is the same as used for the physical target. Note that with virtual target systems, target properties like memory-management units, supervisor and hypervisor modes, endianness, word size, and floating-point precision are simulated and visible. To allow software to run completely unmodified, the model has to model the properties and behavior of the system in a way that corresponds to how the hardware works, at the level where the software talks to the hardware.

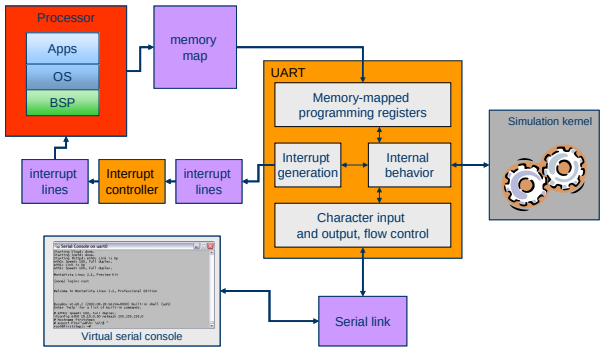
At the hardware/software boundary, the devices in the system show up as registers at certain locations in the processor address space. Device drivers in the software read and write these addresses to make the devices perform. A **register map** (also called the device front end) can be very simple like a few bytes of registers used to control an NS16550 serial port. It can be spread out across the memory map of the processor like a PCI device which is mapped into both configuration space and memory space. The register map can also be very large, like that of a Freescale MPC 8260 CPM which has thousands of registers.

Typically there will also be a **back-end** part of device, where interactions with the environment are handled. For example, a model of the buttons on a watch has to provide some means to "press" the buttons. A screen device needs to display what is being drawn on the host machine screen, and a network device needs to send and receive packets. A multiprocessor system controller will need to interrupt other processors in the system.

Devices also often need to talk to **other devices**, for example to raise interrupts in an interrupt controller or fetch data from memory systems. The front-end memory-map access could be considered a variant of this, but it is usually handled as special cases to improve performance. Similarly, the back-end interface is also access to "other hardware", but since this hardware accesses the external world it is special-cased to provide control and isolation for the simulation.

Behind the register map, the **behavior part** of the device model defines what the device does when it is activated. At a minimum this needs to have enough functionality that the software will work. More functionality can be added to allow fault injection, system behavior logging, and other bonuses. This part of the model **interfaces to the simulator kernel**, in order to things like post events in time, check the current time, initialize connections to other devices, and all other housekeeping tasks. It also orchestrates the model behavior at its other interfaces.

Here is an example of the four interfaces for a serial device:



To build a full-system simulation, you should start with an existing simulation tool or infrastructure. There is no value in reinventing the fundamental wheels of such systems, and using an existing solution also provides an existing library of component models that can speed the time to a complete model, Vendors like ARM, CoWare, Synopsys/Virtio, VaST, and Virtutech offer tools to create virtual systems appropriate for software development (and sometimes hardware development). For those who like open-source, the QEMU project offers a range of targets from which to start developing.

Note that typical desktop/server virtualization solutions like VmWare, Parallels and Xen create a virtual copy of the hardware of the underlying machine. They are thus not full-system simulators in this sense. They do not provide the ability to have a different type of processor (Power on x86), a different variant (Pentium instruction set on a Core 2), or a different clock speed. Not to mention custom hardware like particular Ethernet cards and home-grown ASICs and FPGAs.

Performance Optimization of Full-System Simulation

The speed at which the simulator can execute target code is the key to enabling the use of simulation for serious software work. As shown in the table below, modern embedded software loads easily require billions of target instructions to be executed for anything interesting to happen.

Workload	Size
Booting Linux 2.4 on a simple StrongARM machine	50 M
Booting a real-time operating system on a PowerPC 440GP SoC	100 M
Booting Linux 2.6 on a single-core MPC8548 processor	1000 M
Booting Linux 2.6 on a dual-core MPC8641D processor	3600 M
Running 10 million Dhrystone iterations on a 1-CPU UltraSPARC machine	4000 M
Running one second of execution in a rack containing 10 boards with one 1 gigahertz processor each.	10000 M

A few key technologies make it possible to reach the speeds of 100s of millions of target instructions per second that are needed in order execute interesting workloads:

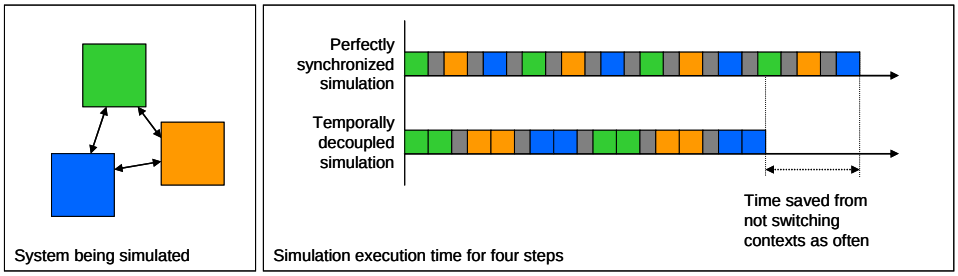
- Binary translation or **just-in-time compilers** to execute the target processor code, rather than lookup-based or interpreter instruction-set simulators. They provide a complete and correct implementation of the programming view of a system, but approximate the timing. Sometimes, they are called “instruction-accurate” simulators to distinguish from more detailed simulators.
- **Transaction-level simulation** abstracts from the cycle-by-cycle, bit-by-bit behavior of the hardware to the events like memory reads and writes which are relevant to the software.
- **Approximations to the timing** of the target system also reduce the amount of detail needed in the simulator, speeding up execution.
- **Temporal decoupling** avoids the overhead of switching contexts between different parts of the simulation on every cycle.

In practice, temporal decoupling and fast processor models are properties of the simulation tool or framework in use. The main work for an end user is to model the devices in an appropriately fast and abstract style, and that is a topic we will cover below.

Temporal Decoupling

Temporal decoupling deserves its own discussion. In computer system simulations, the naïve and obvious implementation technique is to call all devices and processors in a simulation on each clock cycle, and ask them to perform their work for that cycle. This naturally preserves parallel semantics and makes it easy to model the detailed evolution of concurrent hardware units that synchronize on clocks in the actual hardware.

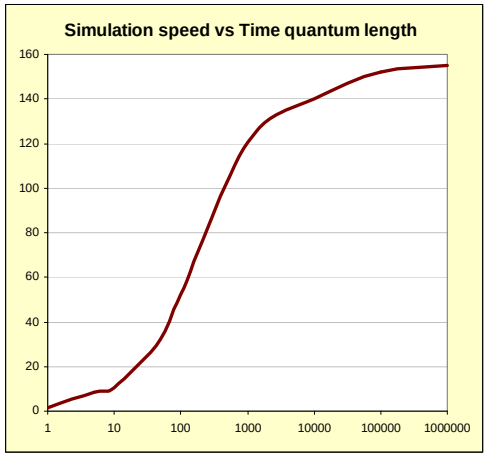
In the context of full-system simulations for embedded software development, such tight synchronization is not necessary. Most of the time, simulation parts will not interact on every cycle, and this can be exploited to let a particular part of the simulation run ahead of other parts of the simulation for a short while. This technique is called **temporal decoupling**, and its consequence is that different parts of a system will have slightly different ideas of the current simulation time.



The simple illustration above shows the idea. Here, each part of the simulation runs two steps instead of a single step each time it is invoked. The reduced number of context switches makes the simulation run significantly faster, even without accounting for the efficiencies of enhanced locality. The length of time a simulation part is allowed to run ahead from the other parts is usually known as the **quantum**.

The idea of simulation time in quanta rather than single cycles is analogous to the way in which transactions group several pin-level operations into a single step.

Experience shows that using a fairly large quantum is critical to achieve really high simulation speeds. The diagram below shows a set of measurements results for booting a four-machine network using the Virtutech Simics simulator. Note how simulation performance increases by an order of magnitude as the time quantum increases from around 10 to around 1000 cycles. As far as the software is concerned, such decoupling is no big issue. Using a quantum of 1000 or 10000 cycles is usually not perceptible to the software, and brings enormous benefits in simulation speed.



System Modeling Methodology

System Component Categories

From the simulation perspective, a target system can be broken down into four main categories of components.

- **Processor cores**. The target CPUs running the target system code. Usually provided by your simulation tool or infrastructure. Highly reusable across targets, as the world contains a comparatively small number of processor core variants compared to other types of components.
- **Memory**. RAM, ROM, EEPROM, FLASH, and other memory types that hold code or data. From a functional simulation perspective, most memories are the same. The programming and control codes for FLASH memories have to be modeled obviously, but otherwise memory is a standard component that can be reused. You should not need to model target memory yourself.
- **Interconnections**. These can be the networks between boards discussed above, as well as the connections within a board (I^2C, PCI, etc.), and the internal buses inside SoC (ARM AMBA is the most well-known example). They are also fairly standardized, and you should be able to get models of the interconnections from your simulation tool or infrastructure.
- **Devices**. Everything that does something in a system and which is not a processor, memory, or interconnect. Examples include timers, interrupt controllers, A-D converters, D-A converters, network interfaces, I2C controllers, serial ports, LEDs, displays, media accelerators, pattern matches, table lookup engines, memory controllers

Most of the effort in your modeling of a custom system should be on device modeling, especially once you or your tool has built up a set of components common to the types of system designs you are working on.

Device Modeling Guidelines

Building a device model is pretty hard the first time. You need to understand your simulation programming framework, the device you are modeling, and then turn the relevant behavior into code. There are many detailed decisions to be made in the design. There are some guidelines that have proven useful to structure the design and implementation work:

- Follow the hardware
- Follow the software
- Follow the boot order
- Don’t model unnecessary detail
- Reuse and adapt existing components

Follow the Hardware

Follow the structure of the physical hardware closely in you simulator. The goal is to ensure that the software developed on the simulated system will run on the physical hardware and vice versa. This includes variations of the hardware configuration. Thus, all software-visible functions have to be

accurately represented in the simulation, and the easiest way to ensure this is to design the simulation model using the same component and communications structure as the physical hardware.

The components to use as the basis for understanding and decomposing the hardware system design are typically entire chips for a board design, and function blocks inside a System-on-a-Chip (SoC). From a simulation perspective, an SoC or a board are really equivalent – they both group a number of devices together with a set of processors, and provide interconnects between them. From a practical perspective, a board often contains some SoC devices, and this leads to a recursive process where the board is broken down into SoCs, and the SoCs into devices.

The starting point is thus the layout of a board and the block diagram for an SoC, as presented by the programmer's reference manuals (PRM). An important source is the memory map typically found in the PRM for both boards and SoCs, showing how devices are presented to the core processor(s) in the system.

Note that some components might be addressed indirectly and not have their own slot in the memory map. A common example is serial EEPROMs accessed over I^2C from an I^2C controller. The EEPROM is not visible in the memory map, but it is still accessible to the processor and needs to be considered in the model.

The ultimate goal is to have a list of the *devices* that make up a system and a map on how they are interconnected.

The interconnections between devices also need to be considered in order to have a good breakdown of a system for modeling. Some interconnections are usually invisible to the software, and thus do not need to be modeled. A good example are on-chip device interconnects like AMBA, CoreConnect, and OcEAN used in various SoC designs. These interconnects are implemented using complex crossbar technology and bus arbitration which do not matter to most software and should be ignored to enhance simulation speed. The best way to abstract the bus system is that it simply transports bytes from one place to another, and that is modeled by mapping devices into the memory map of a processor.

Interconnects that go outside the memory map do need to be modeled explicitly, however. Typical examples are I^2C and Ethernet networks, where it makes sense to model the network moving addressed packets around as an entity in its own right.

Follow the Software

Implementing every available function of a system in order to be complete in following the hardware is usually not necessary in order to run a particular software load. Instead, we should only implement the functions actually used by the software which we want to run. This is commonly the case with integrated chips and SoC devices which contain more functions than are used in any particular case.

When implementing models of new hardware, you should try to target some particular use case initially, and then broadening to other use cases. From a project planning perspective, this means deciding on a pilot software group which is the first to receive the virtual model and work to fulfill their requirements first.

For example, in a particular project I was involved with we needed to model the Freescale MPC8548 SoC. The MPC8548 ha a rich set of external connections such as PCI express, RapidIO, Ethernet, I2C, MDIO, and others. In this particular case, the RapidIO functionality of the 8548 was not used, and thus that function could be left out from the initial modeling effort for the MPC8548. When other systems appeared that used RapidIO, the function was added.

Another example is the Marvell MV64360 integrated system controller. This controller contains a memory controller for a PowerPC processor, along with some generally useful functions like PCI, serial ports, and Ethernet. Many boards using this controller do not use the built-in Ethernet port, but instead they use an external Ethernet chip connected over PCI. In such a case, the built-in Ethernet controller does not need to be included in the model of the board.

Sometimes, the software explicitly turns off some functions or devices on a chip, and in such cases one or more control registers have to be implemented that accept the "off" setting, and give a warning if any other status is written.

It is also good practice to include registers corresponding to unimplemented functionality in the model. Such registers should simply log a warning when they are accessed. Compared to leaving them out altogether, this explicitly documents design assumptions and modeling decisions in the source code. It also provides an obvious indication when the assumptions are violated.

Within a device, only the *functionality which is actually used by the software* should be modeled. This typically means focusing on a few operation modes or functions of a device, and leaving the rest unimplemented (but explicitly so, as discussed below). Often, modeling starts with a completely unimplemented device, looking at how the software interacts with the device to determine what actually needs to be implemented.

Note that an effect of this style of modeling is that even though a device model exists, it might not fulfill the requirements of use in a different system from the one which it was developed for. As time goes on, a device typically gains functionality as it is subject to different uses from different target system configurations and software stacks.

Follow the boot order

In large and complex target systems containing multiple boards (or subsystems), it is necessary to decide the implementation order of the boards. To get to a working model that can run some meaningful software as quickly as possible, the best strategy is to follow the boot order of the system. This means that the first board or subsystem to model is the one that takes charge upon power-on of the system, and that other system components are modeled later.

This is especially relevant when modeling new hardware targets, since the software groups will want to start working with the system boot first. That part is the most hardware-dependent and thus the one for which access to virtual hardware is the most pressing.

Don’t model unnecessary detail

A good model implements the *what* and not the *how* of device functionality. The goal is to match the specification of the functionality of a device, and not the precise implementation details of the hardware.

It is easy to fall into the trap of modeling detailed aspects of the hardware that are invisible to the software. The execution overhead of modeling in too much detail can significantly slow the simulation. A simple example is a counter that counts down on each clock cycle and interrupts when it gets to zero. This should not be modeled by calling the timer model each clock cycle. Note that the counter is only visible to the software when it is explicitly read. A better implementation is thus for the model to register a call-back with the simulation kernel at its expiration time. If the software reads the register before this point, the model has to work out what would be in the register at that point by looking at the current simulation time. Such a model runs much faster than the detailed counter and is indistinguishable from it as far as the software is concerned.

DMA controllers are another example. DMA is properly modeled by moving the entire block of memory concerned at once, and delay notification to the processor (or other requesting device) until the time when the full transfer would have completed on the hardware.

Abstraction can also manifest itself by making entire devices into dummies. For example, a memory controller might have a large number of configuration registers and reporting registers for factors like DDR latencies, number of banks open, timing adjustments, and similar low-level issues. The effects of these are not relevant to a timing-abstracted fast simulation, and thus they can be modeled as a set of dummy registers that report sensible values to the software but writes to which have no effect on the simulation.

A nice side-effect functional modeling is that it makes it easy to reuse models of device functions across multiple implementations of the functionality. As an example, the standard PC architecture contains a cascaded 8259 interrupt controller. Over time, the hardware implementation of the 8259 has changed from being a separate chip to becoming a small part of modern south bridges like the Intel 6300ESB. But despite this change in implementation, the same model can be used, since the functionality exposed to the software is the same.

Note that you can always add details later to a model if you discover that the initial modeling was too abstract. It is about classic software-engineering methodology: first make it work, then make it fancy.

Reuse and Adapt Existing Components

Once a system has been analyzed and its devices and interconnections listed, it is time to start implementing all the required devices. At this point, reusing existing device models is very important to shorten the modeling time. The existence of useful models is important in selecting a simulation framework to use for a particular project.

Sometimes, the precise devices are not available in the library, but there are similar devices. In this case, the existing device can be used as a starting point and adapted and extended to create a model of a new device. Such adaptations of existing devices typically follow the path of hardware designers as they design successive generations of products.

Adapting a device model involves either adding or removing registers, depending on whether we are moving to a less capable or more capable device. It is also commonly the case that some details in the memory map of the device change. Thus, the work of adapting a device starts with comparing the programmer's manuals for the old and new device, and determining the following:

- Identical registers—for an adaptation to be worthwhile, most registers should fall in this category.
- Superfluous registers—functions in the existing model that are not found in the new device. These have to be deleted.
- Missing registers—have to be added to the new device model.
- Different registers—registers with the same function or name, but where the bit layout is different between the old and new device.
- Differences in register layout—the offsets at which various registers are mapped in the device memory map are different.
- Differences in the number of functions—some devices contain repeats of the same functionality, and the difference between devices is the number of functions implemented. For example, a different number of DMA channels or Ethernet controller ports. In this case, simply changing a parameter in the device model may be sufficient.

If there are too many differences, it may be more expedient and safer to implement the new device from scratch. As in all software development, doing too many changes to a model might be more work to get correct than to implement the complete functionality from scratch (maybe borrowing some key source code from the old device model).

Overall Flow

In summary, here are the steps to take when modeling a new system for full-system simulation.

- Map the system, by collecting and reading design specifications, programmers' reference manuals, and other relevant documents. This creates a list of devices and processors that make up the system.
- Based on an analysis of the foreseen system usage, make a preliminary decision on the necessary level of modeling of each device. Can it be ignored, stubbed, or does it need to be fully implemented?
- Reuse existing device models and processor models from the library. The library makes it faster to produce an initial model, since models for many common standard parts already exist. Reuse often indicates adapting an existing model of a similar device, which is much faster then writing a new model from scratch.
- Create initial models of any remaining devices using the Device Modeling Language (DML). Ignore as much functionality as possible initially to quickly get to a basic model.
- Test the new system model. If there is software available, test with the software, and add any missing functionality or devices required by the software. For new devices where no software yet exists, create independent test cases that exercise the specification of the software-hardware interface.
- Iterate until the model runs the available software or passes all defined test cases.

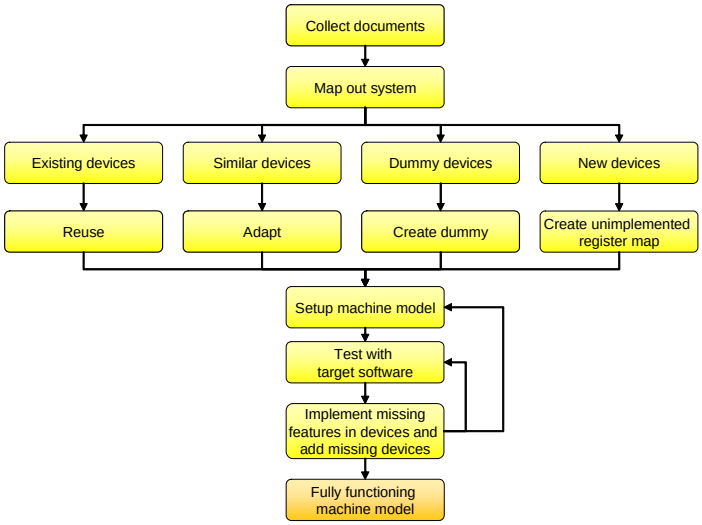
This methodology is a classic iterative software-development methodology, where you test the model (which is the software being developed) early and often in order to explore the precise requirements. Historically, this methodology had gone by many names, from spiral model to agile methods and test-driven development.

The goal is to achieve a model which runs the required software, but that achieves this without implementing unnecessary features and unused functions. Over time, functionality can be added to the model.

Often, it is possible to start using a virtual system almost immediately after starting to develop the model. Even a basic system that does not yet contain all components can be used to get development started. For example, a boot loader typically requires less virtual hardware to be in place than a full

operating system port. Over time, more devices will be added to the virtual system, and it will evolve towards the final model.

The flow chart below summarizes the typical development flow:



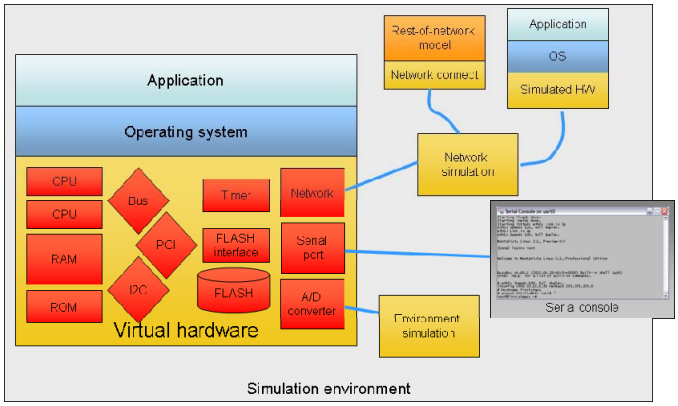
Putting the Pieces Together

So now we know how to simulate an embedded computer and run its software, and how to simulate the environment, user interface, and network in which it operates. What remains is to tie the various pieces together so that the embedded software can indeed run on the virtual computer board, sense and control the virtual environment, obtain virtual user input, and communicate on the virtual network.

To achieve this, we have provided devices on the virtual computer board corresponding to the interfaces of the physical computer hardware. For the connection to the environment, this typically means modeling the processor-facing programming interface of the analog-to-digital and digital-to-analog converters and digital I/O lines. The interface models also have an environment-facing side which is connected to the simulation of the environment, in order to read reasonable values and provide actuation data.

In the same way, networks will be connected to the virtual computer boards using a model of some network interface device. The network simulation can connect the virtual system to other virtual systems, to a rest-of-network model, or some other simulated or real system as discussed above. User interfaces will be represented in the virtual system by a model of the programming interface of a user-interface device like a serial port, LCD controller, keypad interface, LED, or other devices. That device model will in turn connect to a user interface simulator so that the programmer can interact with the system.

The picture below illustrates a typical setup with a serial port, network, and environment connected to a full-system-simulator virtual system.



It is worth noting that simulation of the computer part of an embedded system is useful regardless of the degree of custom hardware used. While much tool attention is focused on hardware designers and users of custom system-on-chip devices, simulation solutions and virtual hardware platforms are just as useful for systems built using standard parts. The value of a virtual computer board and simulated

system to the software developer does not really depend on whether the processor chip is designed in-house or not, it comes from the use of simulation and virtual systems as a methodology to make software development faster and better. Even when hardware is readily available or even in legacy state, simulation solutions bring benefits for debugging and testing that physical hardware cannot match.

Real Time and Simulation?

A common question is the scope of application of full-system simulation, especially with reference to timing-sensitive real-time systems or firmware operating very closely with the hardware. The idea of abstracting the timing of a system is intuitively scary to most people. In practice, however, this turns out not to be a very big deal. Most of the bugs in a software system today tend to be “dumb” bugs like null pointers, missing locks around shared variables, bad priorities for tasks, interrupt service routines that take too long, algorithms which have bad scalability, etc.

Also, most of the code in a modern embedded system is at a higher level than direct hardware interaction. There is logging functions, operations and maintenance, user interfaces, games, remote code update servers, databases, and functions which are timing-critical in the traditional real-time sense. For such code, there is little value in emulating the precise hardware timing. Since processors today use caches, speculative execution, and branch prediction, and other hardware devices like DDR SDRAM also have very variable timing, software has to work over a large range of hardware timings.

Even in the most timing-sensitive control systems, only some 5% to 20% of the code is really that timing-critical. And to be quite honest, that timing has to be validated on physical hardware anyway – no simulation is going to perfectly accurately depict what is going on in the analog domain on a custom board.

This is very different from how things used to be: when I wrote my first assembly programs back in the early 1980s, I knew the instruction timing of every Z80 instruction by heart, as well as the response times of my simple ZX Spectrum computer hardware. Optimization meant shaving cycles by poring over assembly code written by hand. How different from today, where each assembly instructions can vary its execution time by a factor of 100 depending on your luck with the processor pipeline and cache system. Also, in those days you could depend on the hardware never changing. Today, we have realized that software will live for far longer than the hardware, and will run on a variety of different hardware platforms with differences in timing and even device sets. Optimization rather means making sure algorithms scale up with larger workloads, that software is using appropriate system calls, and keeping control over memory usage. In a distributed system, it is about balancing loads across boards. For this kind of work, the coarse-grained timing detail provided by a full-system simulator is sufficient.

Remember, it is better to run *all* of a workload at a sufficient level of detail rather than a small part of the software at a very high level of detail.

Summary

Simulation is a very powerful technique for engineering embedded systems in general, and developing the software component in particular. Depending on where the risk and expense lies in system development, it typically makes sense to build simulators for one or more system parts. Experience indicates that using simulation makes it possible to develop more robust embedded systems faster, letting hardware and software development schedules overlap and providing better debug and testing facilities for software. There are many robust software packages available in the market today that help you construct system simulations, and with powerful PCs available very cheaply, there is no good reason not to investigate simulation for your current or next project.