SAND2011-3991C

Understanding the Complexity of the Software in a Secure Sensor

Troy Ross - Sandia National Laboratories[*]

June 8, 2011

Abstract

Complex sensor platforms are used in safeguards applications to provide reliable identification and monitoring of items of interest. The complexity of the software that controls a sensor platform is rarely understood by non programmers. Unfortunately, this lack of comprehension leads to misjudgments regarding the relative complexity and therefore the amount of effort required in developing a new system or modifying an existing system. This paper will explore general areas in which software complexity lurks and describe in a concise and easy to understand manner the effects of this complexity on the process involved in designing and implementing a sensor platform. This paper will also attempt to associate certain types of requirements and their contributions to the overall amount of effort required to implement a system. The result of this work should assist non programmers in carefully considering their requirements and help them gain insight into the potential pitfalls associated with their implementation.

Troy Ross has extensive experience developing software for embedded seals used in safeguards applications for Sandia National Laboratories. He is the primary author of the software used in several variations of the Secure

[*]Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sensor Platform (SSP) as well as the Enhanced Data Authentication System (EDAS).

Introduction

In nuclear materials management, safeguards, and treaty verification sensors are everywhere. Sensor platforms are incorporated everywhere from the load cells measuring the mass of containers to the fiber optic sealing devices monitoring the integrity of a weapons container. Sensors can vary in complexity from the simplest devices that report their output to analog gauges, to complex devices that require racks of equipment to operate. People who utilize these sensors tend to only recognize the software and hardware, which they interact with. The sensors themselves run embedded software that is responsible for activities that go far beyond the user interaction. It is difficult to conceive of the multifaceted considerations and structures that define embedded software development and the concerns of software itself without understanding the source code and the hardware it is applied to. Programmers also are faced with the difficulty of explaining their work to each other and they often rely upon the power of certain metaphors to convey meaning beyond simple descriptions. Software metaphors which are derived from architecture and construction are often used in software development. This paper will further the reader's understanding of the development of embedded software by comparing some of the concerns which arise when constructing a building to similar concerns which arise when constructing software for an embedded sensor platforms. In addition, where relevant, it will highlight the effects that certain requirements have upon the implementation of embedded software.

Location

Before anything can be built a fundamental question must be answered, where will the building be constructed? Will it be within city limits where certain services are readily available such as water, waste, gas and electricity or will it be built in the countryside where a well has to be drilled for water, a septic system has to be installed for waste disposal, gas has to be delivered, and solar panels need to be installed to generate electricity?

In software, operating systems provide services to programs like municipalities supply services to homes and businesses. Without an operating system these services have to be created within the software itself.

One of the services provided by an operating system is task scheduling. Task scheduling is the process of making sure all tasks are completed in the order of their relative priority. Similar to coordinating all of the individual construction tasks that take place on a construction site such as laying a foundation, framing walls, or installing a roof. Having these services provided is like having a construction manager who handles all of the coordination.

In embedded applications task scheduling is a very important operation, especially for a device designed to sense its environment. Difficulty arises when trying to guarantee that multiple tasks will occur within a certain time limit. Ideally all sensor monitoring activities could happen in parallel and would be allowed as much time as required for them to complete. Unfortunately, sensors are usually not able to operate in a parallel and they must compete for processor time and interrupt each other based upon their relative priorities. For example, a sensor is required to monitor a fiber optic cable every 200ms$\pm$10ms, and in addition it is required to report the state of the cable once a minute over a RF link. The sensor takes 30ms to monitor the cable and takes 200ms to report the cables status over RF. Given these constraints it is impossible to simply allow the sensor to complete its RF communications without monitoring the state of the fiber sometime during RF communications. Regretfully, there are times during RF communications that the communications cannot be interrupted because doing so would cause data corruption, so therefore the start of the RF communications must be delayed just enough so that it can be interrupted during outside of its critical uninterruptable operations. Operating systems which provide scheduling with constraints similar to these are called real time operating systems. Without an operating system, this must be planned out before hand by the programmer and as the number of tasks increases this management becomes more difficult.

In embedded systems or systems destined for sensitive deployment where their every aspect will be inspected, operating systems are frowned upon because with the multitude of services that they provide they also bring with them a significant amount of extraneous functionality. No software requires all of the services that an operating system provides and the services that are not used cannot be easily removed from the operating system. Because of this, often authentication requirements specify that an operating system shall not be used.

Now if the requirements are eased and instead of stating that the fiber optic cable has to be monitored every 200ms±10ms it was restated that the fiber has to be monitored 5 times a second with the maximum time between measurements to be no greater than 300ms. The task scheduling also becomes easier. Constraints which are more lax can be handled with simple task loops which schedule tasks in a simple round robin manner and require no operating system to accomplish the schedule.

Functionality

If an architect designed a building with the understanding that it would be expanded in the future much of the infrastructure may already be in place to accommodate an expansion. If the expansion never occurs, the cost of the extra work and materials utilized to facilitate the expansion will go to waste. Building a building is a very deliberate process because the cost of materials is a major part of the overall cost of building a structure and buildings are meticulously planned before they are built. When building software for an embedded system the software structures are built out of bits, which cost nothing and therefore time is the only cost. Because of this, the design of software is malleable and changes often over time. While an overall architectural plan is in place for an embedded software system, the individual parts of the system are building separately as modules, which individually start out as very simple proof of concept or function implementations. These individual modules are then combined piece by piece until they form the final system. The structures required to support the combination of these modules constitutes the system architecture of the embedded system. This structure begins as minimally as possible, only just enough to support a single module. The very first structures developed are the equivalents of straw huts barely held together, whose purpose is only to provide a place to store data collected from a sensor. Eventually, these preliminary structures are torn apart and rebuilt into more and more substantial structures. These increasingly complex structures not only store the information, but also communicate the data collected to the outside world, analyze the status of the sensor platform, and coordinate the sensor platforms activities.

Adding new sensors to a platform is like adding a new room or even a new floor to a building. Many times adding a room is easy, simply create a new entry way and add three walls and a roof. Other rooms require significant modifications to the existing structure such as moving the associated plumbing, electrical and ventilation. Adding another floor to an existing structure may be impossible without significantly modifying the support structure of the entire building.

The collection of sensors or capabilities that are first specified influences the initial design and infrastructure that is built to support the sensors and related reporting processes. When specifications change many of these changes can be made easily because sufficient support already exists to accommodate the overhead of an additional sensor or reporting process. Sometimes the computing device chosen for the sensor platform is simply not capable of handling the additional load because of limitations in memory, power, or I/O.

Adding capabilities to a sensor can be relatively easy or hard based upon the existing structure of the software before the sensor was added. Many sensors which have binary outputs such as switches are always easy to incorporate as long as I/O channels are available on a device. Moving up in complexity are analog inputs which require analog to digital converters and related procedures to turn on and off the sensing devices. After analog inputs have been digitized some interpretation of their value is required to enable the calibration of their values to standard units. All of these types of measurements are relatively easy, but how a platform begins to interpret these measurements becomes complicated quickly as sensor platforms begins to have required actions based upon ranges of measurements from analog inputs. Just detecting the change in analog inputs requires accounting for measurement error, which may be greater than the change a sensor is trying to detect. All of this logic is in addition to any software that has already been written and the additional memory space required may not be available on a device. Such limitations may induce the modification of other software subsystems to enhance efficiency and the reuse of memory or other limited resources.

Integrated sensors connected through standard buses such as I^2C and SPI are more complex than analog inputs. These sensors have specialized protocols, which must be implemented in the form of drivers on a sensor platform. Support for buses is generally available either through a library provided by a chip vendor or the operating system, so a great deal of effort is not necessary to support these buses. The protocols to interact with the devices must be written and this effort is well documented, but requires detailed testing. Support for buses does not provide a means to handle data from multiple devices connected to the same bus. This additional infrastructure and coordination becomes more complex as more devices are added to the same bus. Finally there are sensors, which do not use standard buses and have custom protocols; these types of devices require significant amounts of work to integrate.

Longevity

Building techniques have been known for centuries on how to construct a building so that it does not fail. Almost any building can stay intact for a century as long as regular maintenance is performed. If maintenance is not preformed and problems are not addressed quickly they can lead to complete structural failure. A burst pipe can easily cause enormous amounts of damage if the flow of water is not quickly turned off. Software can be designed to be fault tolerant and only fail given catastrophic circumstances. For embedded systems the maintenance has to be taken care of by the software itself. The software has to determine when a device or sensor is no longer functioning, and it then has to take actions to either repair the malfunctioning device or permanently turn the device off and exclude it from future operations. There are a variety of techniques to determine if a device is no longer working properly. The first technique is to establish the expected amount of time it should take a device to report back and if the device significantly exceeds that amount of time assume the device is malfunctioning. If such a technique is not implemented, it is possible for a system to wait forever for an answer from a sensor, which is no longer functioning. Another technique is to request the status of a sensor directly through its command interface. Finally, checking the plausibility of the data coming from a sensor will indicate if there is a problem with the sensor. Once it is determined that a sensor is malfunctioning steps must be taken to correct the problem which could be resetting the communications link with the sensor, or the sensor itself. If correcting the problem does not work the sensor has to be excluded from routine operations and its failure reported. If the sensor platform itself is malfunctioning it also has to have a means of detecting its malfunction. This is accomplished by configuring a watch dog timer which resets the platform if it does not reset the timer quickly enough. Requirements for device longevity increase the amount of faults that the device must recover from. Longevity means conserving limited power resources as well as handing a multitude of abnormal situations.

Security

The traditional approach to enhance security on a building is to install locks, reinforced doors, and other protective structures. On the inside of a building the most valuable property is stored in safes or in the case of a bank a vault. Some of these security features can simply be added on after the home is constructed such as more secure locks and doors. While others like vaults are built and installed before anything else is constructed. For security of an embedded system to be effective it must be integral to the design. Cryptography is used extensively throughout embedded designs to obfuscate and to validate information. The algorithms used in designs are mostly available as modules from various vendors that can be integrated easily into a design. The two types of cryptographic algorithms available are symmetric, which means shared key and asymmetric, which means two separate keys. Asymmetric algorithms ease the difficulty of key management and therefore are often made into requirements. Unfortunately, asymmetric algorithms require significantly more power and time to compute when compared to symmetric algorithms; therefore if asymmetric algorithms are required, the frequency of communications or the life of the device is significantly affected. Related to the choice of cryptographic algorithms is the issue of key management. Protecting keys in a device is a difficult problem because once an adversary has access to a fielded sensor platform they have the ability to attempt to extract the secret cryptographic keys that reside in the memory of the device. For embedded software, the first defense is to detect the attempt at unauthorized access to the device. If an attempt is detected, the keys stored in the device are overwritten. Regretfully, the memory on a computer can retain physical evidence of what was previously written to it even after it has been deleted, due to the "Data Remanence Affect[1]." To counter this affect the keys themselves must be periodically moved and their previous location overwritten with information which will negate the affect. Keys are also copied when used in cryptographic operations these copies must also be treated in the same manner as the primary location where the keys were stored.

The smallest defect or oversight can compromise the security of a building.

In a heist movie it is a cliché when, the unprotected ventilation shaft allows a thief to infiltrate the bank in the middle of the night, while completely avoiding the reinforced entry doors and the idle guards watching a video of all known entryways. In embedded software entry ways and ventilation shafts abound in the form of communications and the services that process those communications. The front doors of software where most communications takes place tend to be well protected in that they only accept a specific set of commands and command parameters. Often there are signal buses internal to an embedded system that are more accepting because of the assumption that an internal bus will never be used in an attempt to compromise the system. A command that gives false information with regard to the size of its contents may overwhelm a system and the system could 'fail open,' which could compromise system security. Tools are available which look for many of these vulnerabilities in software, but often they do not understand some of the compiles used in embedded systems or like a building inspector they find certain deficiencies, but overlook others. Therefore it falls upon the software developer to explore potential vulnerabilities and create mechanisms which will prevent compromise when creating or modifying communications interfaces.

Conclusion

The issues raised in this paper are only a sampling of the concerns that affect the process of developing software for an embedded sensor platform. Several examples were given: timing requirements and their effect upon the selection of an operating system, additional sensors and the software support infrastructure changes they incur, sensor faults and the steps to mitigate or correct them, and memory remenence and its influence upon key management procedures. Insights into issues such as these are important to have before specifying requirements or planning for their implementation. Collectively the concepts presented in this paper can provide important knowledge regarding the impact requirements have on the overall complexity of an embedded sensor implementation.

References

[1] Peter Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.