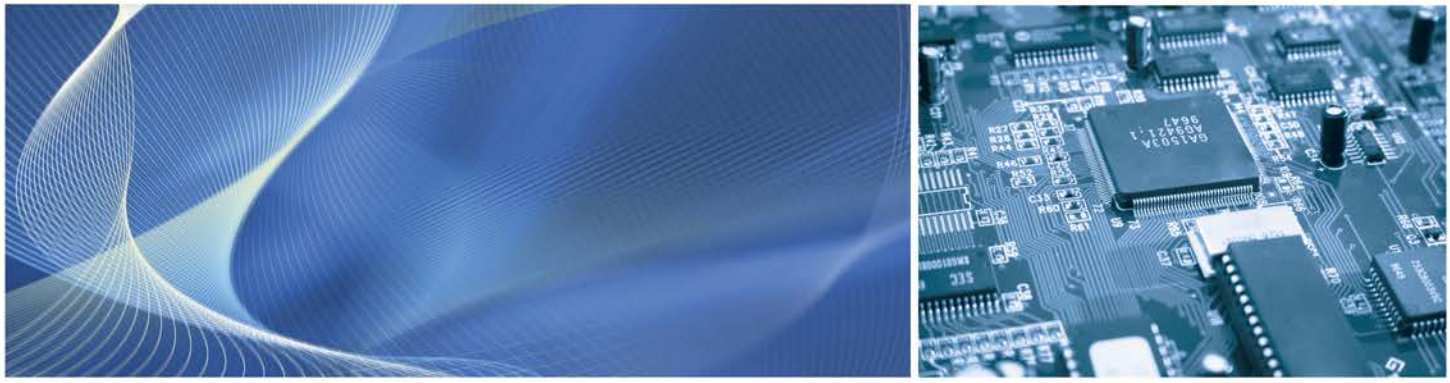
EmbeddedMarket
forecasters

WHITE PAPER



www.embeddedforecast.com

Essential Technologies to Enhance Design Efforts and Design Outcomes for IoT Developers

Jerry Krasner, PhD., Chief Analyst

September 2014

Making IoT Developments Easier to Design and Deploy, Less Costly with Fewer behind Schedule Completions

Jerry Krasner, Ph.D., MBA
Dolores A Krasner

February 2015

Table of Contents:

Regarding the Data in this report

The data that is referred to in this report is *statistically accurate* and *authentic* and is based on:

- A statistically generated comprehensive and detailed survey of embedded developers and managers who reported on their design results (number of developers per project, vertical market of their design, time to market, percent of designs completed behind schedule or cancelled, closeness of final design outcomes to pre-design expectations, testing outcomes, etc.), the tools they used (development, modeling, Java, Eclipse, and other development tools), their choice of OS, IDE, communication middleware, processors used as well as where they go to learn about new products, tools and concepts.

- An EMF Dashboard – a unique tool that allows the user to simultaneously compare similar products (vendors can do competitive comparative analysis); that marketing executives can use for sales promo and strategic planning; that allows developers beginning a project to compare the experiences of hundreds of fellow developers that undertook similar projects to gain insights before making a commitment; and that allows CFOs and senior managers to look at what tools and processes resulted in the greatest cost savings.

For the interested reader, the following link demonstrates the power of the Dashboard and how we used it in developing the data that is presented herein:

http://www.embeddedforecast.com/EMF_DashboardIntro/EMF_DashboardIntro.html

Introduction

I remember as a teenager in the mid-50s watching programs that forecast the "kitchen of the future" and predicted flying cars and exotic neighborhoods. Most of the technology, other than flying cars (we did have amphibious cars that could drive in the water or on roads), was within reach, but those forecasts never came to be. The Latin slogan, *Exeta Acta non Probat* – "not all that is possible actually happens" – is the driving force here. Certainly, the Internet was not foreseen at that time, or the ubiquitous nature of semiconductor technology and the advances in embedded software. What made these forecasts misread the future was that there wasn't a financial structure that supported the availability (nor the desirability) of such possible products.

Today, we know that current technology can enable and connect billions of devices, which can then be accessed from anywhere in the world to enable smarter homes and cities. Small sensors can be attached to remote and dangerous sites (e.g., radioactive) to monitor pressures and radiation levels, to roads to measure deterioration, and these same devices can be applied to agricultural applications and building management, as well as to the "smart home".

All of this is exciting – but putting it all together across different networks, different OSes, and on users' devices requires that the technology can be monetized to the advantage of users, developers, and those responsible for deployment and maintenance.

In the late 1990's the "smart home" promised the user that he/she could program the mirror in the bathroom to present the latest stock market results with commentaries, breaking news and what ever was of interest to each member of the family so that they could bathe, brush and comb without missing a beat. So what happened to the folks that designed the "smart mirror"?

Well, they went broke. The marketplace did demand such access to information but it is delivered free on your smart phone or Tablet.

Do we really need to remotely access our thermostats, open and close our garage doors, remotely unlock our doors and watch our kids come home from school while we are at work? Clever at it is do we really want to pay for these capabilities?

When we leave for our lake house in the mountains of New Hampshire, I turn down the thermostat at home and when I reach NH I turn up the thermostat there. I have better things to spend my money on than this. When I pull in the driveway I hit the remote button to open the garage door – and I close it the same way when I leave.

Personally I wouldn't want to be watching the front door waiting for the kids to come home. What if they come home when I'm not watching? The kids can text me when they come home and I can tell from their GPS where they were when they texted me. Worse yet, if hackers access my home system, criminals might learn my habits of coming and going and be able to unlock my doors and bypass my security systems.

My point in looking at the IoT and the capabilities that are offered me is that I choose how I spend my money and whether or not these capabilities are of sufficient interest to invest in the technology.

In short, the IoT will grow where the technology can be monetized. If it doesn't make money, it goes into the dust bin of past failures.

Let's look at the situation from the viewpoint of the developer.

To do this we need to look at current technologies that provide an integrated development environment that enable developers to get their product to market faster and at lower cost – and to enable rapid modifications to existing products to address the requirements of new end users.

The prospects of what IoT as a critical part of our society can do to benefit our economy and way of life is exciting to contemplate, however the ***chore of getting such technologies to market so as to achieve a competitive financial advantage requires that developers are provided the necessary tools.***

OEM CFOs will need to rethink cost expenditures and cost containments. Up front costs will prove to be inadequate when the cost of being late to market or needing to remove design features is exorbitant.

RTOS Considerations

Developers that choose an RTOS will need to not only do applications development but will certainly be integrating wireless and communications middleware into their designs. USB will find many applications as well. It will be advantageous to developers if GUI development can also be integrated into the design process.

Vendors that integrate wireless (Bluetooth and WiFi) protocols and USB into their operating system will have a definite advantage over other RTOS vendors. Using DDS for communications middleware will enable networks to expand easily to accommodate growth. Modeling will enable code reuse as well as rapid prototyping and integration of legacy software.

Smaller RTOSes such as ThreadX that have integrated wireless capabilities (with Clarinox), integrated GUI design and comprehensive middleware - in addition to USB – will offer developers a better design environment than the larger and more recognized OS vendors.

Let's be truthful regarding the fallacy that open source software offers a financial advantage over commercial OSes. A cost comparison between a commercial RTOS, Linux and open source shows that developers use of commercial RTOSes will get to market faster and at a much lower cost.

The following Table is based on the results of 615 respondents to a detailed survey of embedded developers and managers who reported on their design results. The data that is referred to herein has a *statistical confidence at the 95% level.* These data included the number of developers per project, vertical market of their design, time to market, percent of designs completed behind schedule or cancelled, closeness of final design outcomes to pre-design expectations, testing outcomes, etc.), the tools they used (development, modeling, Java, Eclipse, and other tools), their choice of OS, IDE, communication middleware, processors used as well as where they go to learn about new products, tools and concepts (www.embeddedforecast.com).

EMF Survey Data All Respondents	**Ind ave**	**ThreadX**	**Open Source**	**Comm Linux**	**Non-Comm Linux**	**Microsoft CE**
Devel time Months - Start to Ship	13.9	12.9	13.2	12.9	15.1	14.3
% behind schedule	47.0%	36.9%	45.1%	47.5%	46.7%	38.1%
Months behind	3.8	3.0	3.6	3.1	4.1	4.0
% cancelled	11.2%	13.4%	10.9%	11.7%	11.2%	11.5%
Months before cancellation	4.4	4.6	4.4	3.6	4.4	4.4
SW Developers/project	14.7	6.8	19.0	17.4	22.3	13.3
Average Developer months/project	204.3	87.7	250.8	224.5	336.7	190.2
Developer months lost to schedule	26.3	7.5	30.8	25.6	42.7	20.3
Developer months lost to cancellation	7.2	4.2	9.1	7.3	11.0	6.7
Total developer months/ project	**237.8**	**99.4**	**290.8**	**257.4**	**390.4**	**217.2**
At $10,000/developer month						
Average developer cost/project	$2,043,300	$877,200	$2,508,000	$2,244,600	$3,367,300	$1,901,900
Average cost to delay	$262,542	$75,276	$308,484	$256,215	$426,978	$202,692
Average cost to cancellation	$72,442	$41,915	$91,124	$73,289	$109,894	$67,298
Total developer cost/project	**$2,378,284**	**$994,391**	**$2,907,608**	**$2,574,104**	**$3,904,173**	**$2,171,890**

Table I: Worldwide Comparative Costs of Development

ThreadX was selected as the comparative commercial RTOS. These results have been repeated year over year. Clearly open source software may be “free” to use but we caution CFOs and financially responsible managers that it is important to consider the “total cost of development” rather than “acquisition costs” when making development decisions. EMF data going back more than 5 years has shown that the development costs of “free” Linux far outweigh the costs of developing with commercially available Linux. Those that follow EMF publications will recognize this. What is less recognized are the comparable costs of commercial RTOSes versus open source developments.

Certainly, not all commercial RTOSes fair well against open source software, but many do. We caution IoT developers (and their managers) to explore comparable costs before making final decisions.

Communications Middleware

Conceptual Considerations

When one thinks about it, there are a number of reasons to expect that commercial middleware provides better ROI than “roll-your-own” (RYO) in-house solutions.

The most fundamental reason is that RYO solutions tend to be designed and implemented based on *initial connectivity requirements* and thus are very brittle when new requirements are introduced. They typically rely on direct, point-to-point connections between communicating applications/components. Often, applications just send and receive data/messages using the ubiquitous *“sockets”* library with TCP as the underlying network protocol. (TCP is used because it is connection-oriented and provides reliable, ordered delivery.)

These designs are *tightly coupled* because each application has to have knowledge of the other applications with which it is communicating and of the application-level protocols. If a sensor (such as a radar) is sending data to a signal processing application, it has to know how to address and communicate with that application, e.g., that it has to establish a connection to port 12345 on signal.mycompany.com and know how to format data for the signal processing application. (Or, the signal processing application has to know to poll for data on port 54321 on sensor.mycompany.com.)

With a connection-oriented architecture, when new applications are introduced old applications have to be updated to know how to communicate with them. Having to update old applications adds cost for development and re-testing/certification. It makes it much more expensive and time consuming to insert new technology into existing systems. This gets worse over time as systems become more and more stovepipe and brittle.

Commercial messaging and integration middleware overcomes this problem by providing *loose coupling*. Applications don’t have direct knowledge of each other. They send and receive data/messages through an abstraction. For example, with publish/subscribe I send data to a “topic” and anyone who subscribes to that topic will receive the data. The publisher doesn’t care who the subscribers are and a subscriber doesn’t care who the publisher is. The middleware does all the work of matching publishers and subscribers and properly routing data. It provides the software analogue of a hardware bus. New applications can be added without affecting existing applications. Thus, systems are much less expensive to maintain over the long run.

Other limitations of RYO addressed by commercial middleware include:

- RYO typically isn't originally designed to accommodate heterogeneous systems, with mixed processor types, operating systems and programming languages. If every application is written in C and running on a 32-bit PowerPC processor everything may be fine. But add a Java application running on 64-bit Intel Architecture, the odds are almost nil that they will interoperate. An internal software layer will have to be created to properly serialize and de-serialize data/messages so they are interpreted correctly across platforms.
- RYO doesn't accommodate disparate Quality of Service requirements – such as persistence/durability, transactions and timing constraints. What happens if the signal processing application goes down? How does it get data that it may have missed?
- RYO doesn't accommodate network disruptions, failures, dynamic or ad hoc environments

Over time, as the above issues are addressed, RYO middleware that might start as a simple application-level protocol over TCP *sockets* often ends up becoming a full blow infrastructure with its own APIs, need for maintenance, QA, documentation, support of internal end-users, porting to new platforms, feature enhancements, etc. All of this costs money.

There is also no ecosystem around RYO, further hurting developer productivity and increasing the amount of custom code that must be developed. RYO is not integrated with tools such as Rhapsody. It is also not integrated with other middleware or COTS applications. A benefit of commercial middleware is that there are standards for middleware-to-middleware interoperability, so, for example, someone using RTI can interoperate with someone using IBM MQ without too much effort. If interoperability with 3rd-party middleware is a requirement for someone using RYO they would have to do much more custom integration work.

Introducing DDS:

Machines are becoming more software intensive, programmable and connected. This trend is accelerating as organizations exploit the Industrial Internet and Internet of Things (IoT). With systems increasingly distributed and open, their underlying communication approach can no longer support the required scalability, interoperability and flexibility.

Connext DDS (offered by RTI – EMF data shows that RTI's DDS offers a lower average cost of development than the DDS industry taken as a whole) products provide a robust and scalable messaging foundation for more intelligent and connected machines. They offer a simple programming model, flexible integration capabilities and open interfaces. This allows software architects and developers to:

- Develop distributed applications faster

- Integrate systems more efficiently
- Maintain and evolve products at a lower cost

High-level publish/subscribe messaging APIs ease distributed application development. To communicate, software components just publish the data they produce and subscribe to the data they consume. Connext DDS automatically routes data between publishers and subscribers of the same data topic. Applications do not have to deal with complex low-level networking details like sockets, addressing, discovery, reliability and serialization.

Due to the complexities of interconnecting between different communications networks, EMF suggests that developers look to DDS providers.

Model Driven/Model Based Development

Introducing Model Driven Development (MDD)

Model Driven Development (MDD) is the latest step of abstraction in writing software applications. Software development has moved from machine language to FORTRAN to C and C++ languages, as well as to Java and to the SQL database. As we examine each step along the development levels of abstraction, we see that the higher levels of abstraction have offered significantly improved productivity and ease-of-writing applications. In such, developers can handle increasingly complex developments without increasing the development work load thereby enabling applications to be developed faster and less costly than with previous techniques. MDD is also called a simulation modeling tool.

MDD separates the model from the code enabling the developers to work on a platform independent model. The auto-code generation capability writes the code according to the underlying OS and processor used which enables rapid prototyping of product subsystems.

For more complex systems, MBSE (Model Based Systems Engineering) adopts the same MDD like approaches to systems of systems, sub systems and systems integration beyond just embedded code.

EMF data has clearly shown that the total cost of development can be orders of magnitude *less* than acquisition costs in appropriate circumstances when MDD is employed:

- On time shipment of product – MDD developments ship faster than comparable developments not using MDD.
- Maintaining the expected performance, systems functionality and features and schedule of the development – Final design results are closer to pre-design expectations for MDD developments.
- Achieving market windows of opportunity – MDD developments not only get to market faster, but they can be easily upgraded to meet new market opportunities by integrating legacy software into new designs and automatically generating and deploying new code.
- Cost of re-design necessitated by changes in available hardware is minimal with MDD.
- Reduced project costs by finding problems earlier, when they are cheaper to fix. Previous research (Defense Systems Management College – 1993) has shown that after having spent just 15% of the development budget, 85% of the costs are already baked in. What costs X to correct in the early stages can cost 10X or more to correct later.

- Cost of in-field support is more effective because support personnel can more clearly understand design models and code versus just source code. MDD minimizes the possibility of product recalls.
- Documentation is automatic. The original developers may have retired but with MDD all of the documentation and interface information is retained. Imagine the cost of upgrade if all of the software apps had to be redone.

The following Table presents a development cost comparison between similar projects that use MDD compared with those that don't.

MDD Worldwide	**North Amer MDD**	**North Amer**	**Europe MDD**	**Europe**	**Asia MDD**	**Asia**
Devel time Months	14	14.3	14.1	14.4	10.4	10.4
% behind schedule	39.4%	39.4%	33.5%	38.3%	33.6%	32.9%
Months behind	4.1	4.4	5.0	4.5	2.6	4.2
% cancelled	9.0%	10.3%	10.1%	8.0%	9.8%	11.3%
Months lost to cancellation	4.9	9.4	5.1	8.6	3.3	18.3
SW Developers/proj	8.0	16.9	9.4	16.9	6.7	16.9
HW Developers/proj	6.4	6.5	3.9	5.3	4.2	10.5
Total project developers	14.4	23.4	13.3	22.2	10.9	27.4
Average Developer months/project	201.6	334.6	187.5	319.7	113.4	285.0
Developer months lost to schedule	23.3	40.6	22.3	38.3	9.5	37.9
Developer months lost to cancellation	6.4	22.7	6.9	15.3	3.5	56.7
Total developer months/ project	231.2	397.8	216.7	373.2	126.4	379.5
At $10,000/developer month						
Average developer cost/project	$2,016,000	$3,346,200	$1,875,300	$3,196,800	$1,133,600	$2,849,600
Average cost to delay	$232,618	$405,662	$222,775	$382,617	$95,222	$378,613
Total developer cost/project	**$2,248,618**	**$3,751,862**	**$2,098,075**	**$3,579,417**	**$1,228,822**	**$3,228,213**
		66.9%		**70.6%**		**162.7%**

Table: Comparisons between Similar Projects using and not using MDD

The Asian data appears to be out of line with the results from Europe and North America. This is due to the fact that Asian projects use more developers per project because the cost per developer is less. However, Table I assigns the cost of developers for all regions at $10,000 per developer per month. The actual Asian cost per developer is much less. We include the data as derived since it is the comparative cost between MDD and non-MDD developments that is important. One would get the same results irrespective of the cost per developer.

Year over year EMF data has also shown that:

- UML and SysML are the most popular graphical representation for simulation-modeling tools for discrete embedded system designs.
- UML, simulation and code generation enables faster design iterations that produce desired performance, functionality and capabilities.
- Using UML, simulation and code generation, design cycles are more predictable and result in faster product shipments with lower project risk.
- UML simulation and code generation contributes significantly to a reduction in design, development and implementation costs.
- The use of simulation-modeling tools by embedded developers has reduced design delays and cancellations.
- The use of simulation-modeling tools by embedded developers has significantly improved the relationship between pre-design expectations and final designs.

EMF data has shown that MDD enhances design outcomes and saves substantial monies. Thus integrating MDD into existing developments need not be disruptive and can provide the following:

1) Effective code reuse (and other artifacts) capability
2) Ability to redeploy applications when underlying hardware is changed
3) Ability to upgrade or include prior designs by importing them into the MDD tool from which they can be treated as an external legacy or be integrated into the model.
4) Interfaces from imported legacy code can be visualized. This is very important when attempting to replace systems components many years later when the original developers may have left the company
5) C and C++ developers can work on the same development each within their own familiar GUI – there is no need for C developers to have to learn to use OO techniques

MDD/MBSE also enables a more efficient development management process. The advantages listed below clearly reflect ongoing cost reductions.

- *More mastery* – Developers can become skilled in one or two processes, but it is difficult to be fully proficient in more. With MDD there is only one process to master.
- *Less downtime* – Developers have fewer training classes to attend and fewer user manuals to consult.
- *More coherent vocabulary* – Development teams with fewer processes have a common language based on those processes.
- *More skills portability* – Developers who move from one project to another can participate more quickly when the new project's processes are like the

prior project. Having fewer processes to choose from increases the chance a developer can transition smoothly.

- *Better focus* – Management focuses on product quality and staff quality, not on process management and churn.
- *Better communication* between designers and developers - This reduces redesign and rework costs.

MDD enables cost savings that aren't achievable using other methodologies. It enables:

- Long term design and upgrades. Legacy developments are easily integrated into new designs
- Code reuse capability – There is no need to rewrite established applications.
- Code portability when underlying hardware is changed and if the OS is changed
- Maintaining documentation
- Ability to capture corporate best practices so newer staff can be on-boarded faster and get up to speed more efficiently

MDD/MBD Suppliers

There are three recognized suppliers of modeling technology. MathWorks' Simulink is a unique dynamic modeling tool (MathWorks calls the methodology MBD – Model Based Development) which is not based on UML. Simulink can be integrated with either PTC's (formerly Atego) Artisan Studio or IBM's Rhapsody.

The two principal UML MDD products are IBM Rational's Rhapsody and PTC's Artisan Studio. The MDD comparative data presented in Table I are primarily based on Rhapsody, Artisan Studio and Simulink. Each of the three principal modeling vendors are established billion dollar companies that offer stability and support to users.

There are two leading, low cost modeling tools and several open source tools available to the market. However, these are not generally considered industrial strength tools and should generally be avoided as they allow the users to design and build embedded applications which are often not correctly constructed or in accordance with the UML/SysML standards. These low cost or free tools are the ultimate example of how starting out cheap will more than likely cost much more in the long term.

Enhancing the Design Process – looking at developmental efficiencies using wireless protocols

Developer's Use of Wireless Protocols

An analysis of data taken from the 2014 EMF Annual Survey of Embedded Developers enables us to compare wireless choices of medical device developers with wireless use across the embedded industry.

The following Tables present developer responses indicating a comparison between medical and other embedded developments for "currently designed-in" and for "plan to use in 2014" developments.

EMF cautions the reader that what developers indicate they "plan to use" frequently does not match what they actually use in the following year. Nonetheless, it is interesting to report on what they anticipate using in 2015.

Currently Designed In	Industry	Medical
Bluetooth Classic V2.1	17.6%	35.4%
RFID	13.4%	29.3%
802.11g	23.7%	28.0%
Zigbee	21.8%	24.4%
802.11b	17.4%	19.5%
HTTP	17.2%	19.5%
Bluetooth Low Energy V4.0	6.5%	18.3%
802.11n	14.5%	17.1%
GSM	13.8%	15.9%
802.11a	14.7%	14.6%
XML	11.1%	14.6%
IrDA	8.4%	13.4%
Proprietary	11.7%	13.4%
3G	18.2%	12.2%
Bluetooth High Speed V3.0	8.0%	12.2%
NFC	6.3%	9.8%
WPA2	9.2%	8.5%
WPA	7.5%	7.3%
WEP	6.7%	6.1%
802.11i	5.9%	4.9%
4G	6.9%	3.7%
CDMA	8.6%	3.7%

Table: Comparative Current Wireless Use

A side by side inspection of Table II shows the preponderance of Bluetooth use (all versions), WiFi, and Zigbee. We report WiFi separately by protocol, and Bluetooth by version, rather than cumulatively whereas ZigBee is reported as cumulative. RFID use was interesting to observe. However, RFID has been used predominantly to control inventory and for surgical use rather than for patient monitoring applications.

Of interest is the use of Zigbee by developers. The frequency of use is comparable between embedded industry use and medical device developments. Yet the 24.4% reported use in the last year is significant, however the “plan to use” shows that ZigBee use is dropping off and both Bluetooth and WiFi continue to bloom.

Plan to use in next 12 months	Industry	Medical
Bluetooth Classic V2.1	14.2%	28.8%
Bluetooth Low Energy V4.0	14.2%	28.8%
802.11g	18.7%	24.7%
802.11n	15.9%	20.5%
802.11b	10.7%	19.2%
RFID	9.2%	17.8%
Zigbee	14.7%	17.8%
802.11a	8.2%	13.7%
HTTP	13.2%	12.3%
Proprietary	9.2%	12.3%
Bluetooth High Speed V3.0	7.0%	11.0%
IrDA	4.5%	9.6%
NFC	5.2%	9.6%
3G	11.4%	8.2%
WPA2	6.5%	8.2%
GSM	7.7%	6.8%
WEP	3.2%	6.8%
XML	7.2%	6.8%
4G	8.0%	4.1%
802.11i	3.7%	2.7%
LTE	4.2%	2.7%
CDMA	4.0%	1.4%

Table: Comparative Anticipated (2015) Wireless Use

Examining Table II we can see from developer responses the drop in anticipated Zigbee use which is consistent with what Clarinox has reported to us.

It is interesting to see that Bluetooth use (and anticipated use), as reported in Figures I and II seem to be consistent. The EMF takeaway is that Bluetooth users (for Classic, Low Energy and High Speed) are satisfied with the performance, cost and associated implementations and will be unlikely to change to another protocol.

RTOS and other vendors be aware – it is a good investment to integrate and support Bluetooth as part of your OS offering as a means to enhance the design experience of your customers and prospects. Bluetooth use among embedded developers is high and enabling developers to easily integrate it into current designs. Clarinox, by offering Bluetooth, WiFi and Zigbee capabilities, has established partnerships with several prominent RTOS vendors.

Table III presents a listing of criteria deemed most important for selecting Bluetooth protocol stack.

Most Important Selection Criteria	Industry	Medical
Microprocessor support	42.5%	42.5%
Reliability	31.5%	40.0%
Compatibility with our development tools	37.0%	37.5%
Availability of source code	37.8%	35.0%
Real time performance	27.6%	35.0%
Acquisition cost	44.1%	32.5%
Quality of support	18.1%	32.5%
Host platform support	22.8%	30.0%
Includes good development tools	21.3%	27.5%
Supports easy porting of existing software	8.7%	20.0%
Memory constraints	13.4%	17.5%
Compatibility with suppliers and vendors	13.4%	15.0%
Must be free (for both development and production)	26.8%	12.5%
Performance on internal evaluation	11.0%	12.5%
Provides device drivers and/or Board Support Packages	15.0%	12.5%
Royalty cost (production licenses)	12.6%	12.5%
Security	10.2%	12.5%
Customer approved or specified	9.4%	10.0%
Must be open source	12.6%	10.0%
Safety certifiable (DO-178B/C, IEC 61508, FDA, etc.)	5.5%	7.5%
Availability of perpetual license	22.0%	5.0%
Availability of professional services (porting, integration, or qualification)	13.4%	5.0%

Table III: Reasons for Selecting a Bluetooth Stack

From Table III we can see that the Bluetooth acquisition cost is less important for medical developers and that they don't expect any freebies. So look out chip providers that include a Bluetooth stack with the processor. Developers want

more flexibility and are willing to pay for it. Also, the availability of a perpetual license is not an issue for medical applications (only 5% report this to be a concern, compared with 20% for the embedded industry).

Issues of importance to medical device developers are realtime performance, quality of support reliability and host platform support. On these points the robust stack offering from Clarinox with support for multiple RTOS vendor products is well aligned to the needs of the medical industry.

EMF asked medical device developers what aspects of their protocol software selection process turned out to be the most disappointing.

Table IV presents their responses which are compared to respondents from the broad embedded industry.

Most Disappointing Aspects of Protocol Software Selection

	Industry	Medical
Expectation of better support	45.8%	50.5%
Better technical solution	45.8%	47.3%
Product integration	40.6%	47.3%
Better price	26.3%	23.1%
Open Source	14.4%	16.5%
Compatibility with customers and suppliers	20.0%	14.3%
Vendor reputation	5.2%	4.4%
Ease of purchasing	7.7%	3.3%
Corporate standardization	8.5%	3.3%

Table IV: Most Disappointing Aspects of Protocol Selection

Wireless - EMF's Takeaway:

Of the short range wireless technologies Bluetooth and WiFi are the most adopted by medical device developers. This gives them both the opportunity to interact with a large number of existing devices and infrastructure.

WiFi is used for higher data rates and longer distance but is more susceptible to co-existence issues. Bluetooth caters to lower data rates/shorter distance (than WiFi higher data rate/longer distance than RFID) but the technology is less affected by large amounts of RF transmission within close proximity and the battery lasts longer.

A key requirement is reliability – and some protocol stack software is more robust than others. Clarinox is the partner of choice of many of the leading RTOS vendors due to the superior robustness and reliability of the Clarinox Bluetooth and WiFi stacks. *By supporting Clarinox stacks into the specific RTOS, the design experience is enhanced for medical device developers and it is easier to achieve better time to market*.

Bluetooth from the module or chip vendor will provide minimal standard functionality – but for anyone that seeks faster pairing, maximized performance and faster data rates, plus a larger range of applications – then a dedicated stack vendor is typically preferred.

What medical device developers need to consider when choosing a wireless technology

- **Range** – Different technologies have different ranges of transmission.
- **Data rate** – How much data needs to be transferred?
- **Battery life** – Is the device battery operated? How long does the battery need to last?
- **Latency** – How quick does the connection need to be?
- **Robustness** – Medical devices cannot afford to require frequent reboot.
- **Wireless technology standards** – Is it important to interact with other devices such as phones or tablets? If so, then a standard technology that is included on those devices should be used.
- **Regulations** – Regulations around the use of RF within some medical environments will be controlled. This level of control may vary among different governing bodies (jurisdictions).
- **Coexistence** – Increased radio frequency “traffic” increases the chance of the RF equivalent of “grid lock”. Some technologies are more affected than others.
- **Security** – privacy and security of data concerns
- **Electromagnetic Compatibility (EMC)** - evaluation requirements for a number of devices such as active implantable cardiovascular devices that provide one or more therapies for bradycardia, tachycardia and cardiac resynchronization

When a Bluetooth provider takes steps to enhance the wireless design process (beyond what the chip manufacturers provide), EMF believes that they should be recognized. Clarinox is one such vendor.

Clarinox watched the trend towards greater design complexity for some time and saw it was not just a problem for them - it is an industry wide issue. So they decided to produce software systems to alleviate the complexity burden for the embedded software engineer - the result is a suite of protocol stacks, middleware, software debug tools and the latest addition, Jannal - Intelligent

Design Environment - all constructed with the aim of helping embedded software engineers with support and effective debug tools.

Jannal, a new generation of Intelligent Design Environment, allows engineers to rapidly develop embedded wireless devices without needing to learn the entire underlying system architecture. Get the full benefit of world leading Clarinox expertise in this comprehensive software suite complemented by the Clarinox Koala EVM. Based upon STM32F407 the Koala board has 2 wireless module interface sockets and provides Ethernet, LCD, Camera, USB OTG, UART/CAN Bus and built-in JTAG support.

Through Jannal the engineer can access the ClarinoxBlue Bluetooth Classic and Bluetooth Low Energy protocol stacks as well as the Wi-Fi WLAN stack complete with WiFi Direct. In fact, the Bluetooth and Wi-Fi stacks run together within the same framework and so help to ease the integration burden if both technologies are to be included in the one design. More information on Jannal is available at http://www.clarinox.com/docs/whitepapers/ClarinoxWhitepaperJannal4wireless.pdf for Koala EVM at http://clarinox.com/index.php?id=415

Why taking the Bluetooth stack provided by chip manufacturers may not always be the best option

- It creates dependencies that are best avoided; if a product is to be in the field for many years end Of Life (EOL) issues must be dealt with. Choosing software that is provided with a chip then means that the software will also need to be changed when the hardware come EOL. This is double trouble. Much better to pick a Bluetooth stack that has an abstraction layer so that the same upper layer software will run despite changes in hardware. Increase quality and performance by choosing an independent stack.
- Other than the dependency on the hardware vendor there is the issue of meeting requirements. What if the required RTOS is not supported? What if the required functionality is not supported? What if the required quality is not provided? Let's face it. Chip vendors are in the numbers game and as such they must cater for the mainstream. But what if you want to create functionality that is beyond mainstream? The choice in these situations is to pick a stack vendor that is independent; specializes in Bluetooth technology and takes the time to provide for new innovative functionality.

USB

Based on the responses to the 2014 EMF Survey of Embedded Developers (864 responders) the following USB user data is presented.

- USB had the lowest average development cost per project ($1.48 million) compared with the industry average ($2.98 million), TCP/IP ($2.53 million), RS232 ($2.42 million) and Ethernet ($2.78 million.

- Significantly more USB applications require IEC 61508 (the primary industry standard for safety) than the industry average

- Regarding characteristics that are most important in making a buying decision, USB developers are significantly more sensitive to “value of tools”, to “ease of use”, and to “speed performance” but not as interested in “compatibility.” The cost of tools is a larger fraction of the project cost, because labor input is lower. Ease of use is important, because there are half as many developers on the project, compared to industry average. Compatibility is not as important, because the project tools do not need to be amortized over multiple products. Speed/performance is important, because (during development) this means less time spent by developers waiting for the tools.

- When asked about what factors were most disappointing after purchasing the product, USB developers are like the industry in almost every way except one: They are significantly more likely to encounter “non-responsive technical support”. This makes sense given that their vendors are normally *not* companies with strong USB expertise.

- USB Developers are significantly more likely to use Embedded Linux, Android, Free RTOS, Windows CE, and Windows XP Embedded; they are significantly less likely to use Green Hills Integrity. This is again easy to understand. Embedded Linux, Free RTOS, Android, Windows CE and Windows XP incorporate USB stacks. The stacks vary in quality.

- Free software is important to more USB developers than the industry – but that it’s only important to one developer in five, leaving ample room for commercial products – particularly those that provide support that doesn’t come with the free stuff.

- USB developers want support for middleware – many USB protocols are part of a larger application involving higher level (USB-independent) functionality. In many cases, it's much harder to integrate middleware with a USB stack than to integrate a USB stack with a given RTOS.

- Cost is more of an issue relative to the industry – again, with the relatively lower labor costs for USB-related projects, this puts pressure on the cost of outsourced components, relative to broader industry practices. However, the vast majority of USB developers do not view "free" as essential. This corresponds to an understanding that support is important.

In summary, it seems clear from the presented data that USB developers are skeptical that they'll get good support from their USB vendor. They therefore demand features and source code access; in the absence of good support, they are more likely to push for a free solution (so that they have budget to pay for the self-support efforts).

The EMF analysis also suggests:

- Data shows that USB developers appear to be unaware of the test resources available to them. Bear in mind that testing can be done in-house for free, or with an independent test lab for something from $1K to $5K. This level of willed blindness suggests that most USB developers are profoundly unfamiliar with the technology they are using; otherwise they would have identified this as a safety and security issue.
- Although 40% of USB developers use internally developed test procedures, only 10% use the USB compliance tests. This means that *at most* one in four of internally developed test procedures are actually testing for standards compliance.
- An embedded host stack that is highly compatible with Windows (such as MCCI's TrueTask USB, which has been sold in high-volume production as a Microsoft-compatible stack) will be <u>much</u> more compatible with the devices in the field than an embedded host stack that has not undergone similar testing
- The data shows that *there is a significant opportunity for RTOS vendors if they can provide well integrated USB support targeting a broad variety of platforms*. They may be able to increase their market share, and capture more revenue.
- A commercially useful solution must support a much broader range of device classes, because a project that needs an obscure device class not supported by a commercial package off-the-shelf is not likely to be willing to subsidize the development or take the schedule risk for developing the new device class.

In selecting a USB stack, developers should be aware of USB options and capabilities. Using an independent provider rather than a chip-based solution (such as MCCI) is a terrific starting point. You will get good insights and have your questions answered if you contact them.

Here's one example of the complexity involved with USB design, and the risks you may face if you try to do it on your own, or try to use a free solution. USB device and host blocks in most SoCs are typically not designed by the silicon vendor. Normally they are purchased as "IP blocks" from an "IP vendor" – the most famous vendors being Synopsys and Cadence Design Systems. Unfortunately, even if the register set appears to be standard, there is room for substantial design variation.

- The SoC vendor will choose a version of the IP block based on many considerations (many unrelated to USB); each version has unique features (and defects).
- The silicon vendor's design team has substantial freedom to configure the IP block when integrating into the final SoC.
- There is substantial glue logic between the SoC core and the USB block, and the SoC design team may introduce optimizations that have substantial software impact.

It's important to choose a USB stack vendor based on experience working with the targeted USB IP block, and also based on their relationship with the IP vendors and the silicon vendor, and their experience working creatively and constructively in a multi-vendor environment.

Things to consider before embarking on a USB development:

- USB is complex – with an installed base of 10 billion devices (and growing) and hundreds of thousands of distinct device types
- USB must be rock solid to keep from interfering with user experience
- Can't easily be built from scratch
- Certification/testing from scratch is nearly impossible
- Efficiency requires reuse of existing USB platform for multiple connectivity requirements (internal, external, cross-product, cross IP)
- Typical freeware is clumsily coded, lacks support, and is difficult to troubleshoot and adapt

Software/hardware Protocol Level Debugging

The 2014 EMF survey reported the mean value of an organization's annual budget for development tools to be $216,239, while the median value for the same question was $16,238. The chart below confirms, while a small number of companies have annual development tool budgets in excess of $1 million, most must live within more modest means.

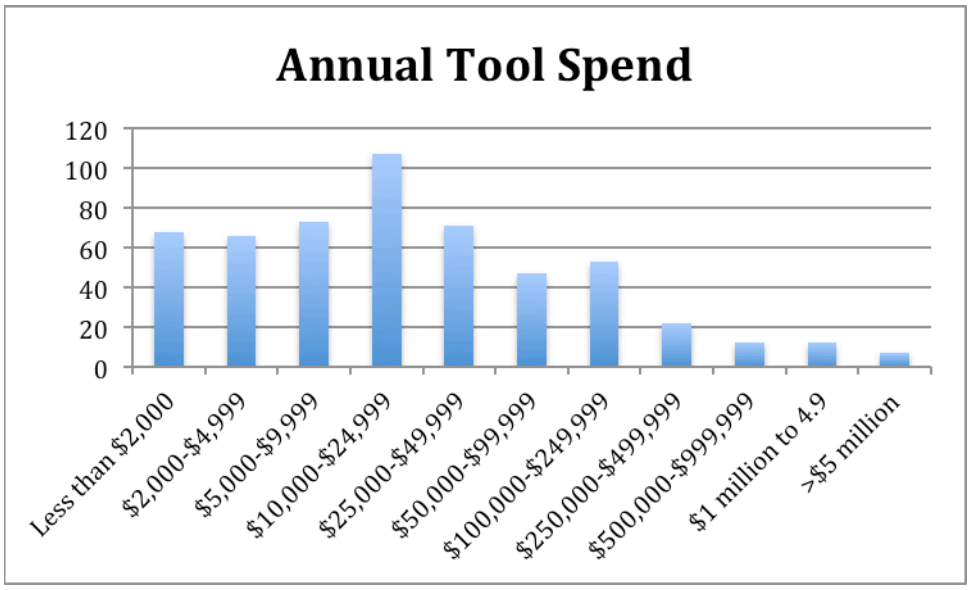


Figure 1 – 2014 EMF Survey, Annual Development Tool Spend

Traditionally, the only option available for companies lacking large capital tool budgets has been to lease those same, cost prohibitive tools. The challenge of squeezing utilization of development and debugging tools (such as high speed oscilloscopes, vector network analyzers, RF sources, etc.) into a short time frame to make cost effective use of rental equipment is difficult, and often results in additional project challenges.

A trend over the past decade has been the deployment of smaller tools which can accomplish much of the functionality of the larger tools, with minimal setup and substantively lower costs. Protocol analyzers passively sniff the bus and report all traffic, providing developers with a protocol layer transactional history that often helps pinpoint issues between host and device. Host adapters enable developers to send/receive messages on a bus. The applications for host adapters include emulation of master/slave behavior in the early prototyping phase, injection of protocol layer errors to test fault tolerance and stability, and general purpose qualification.

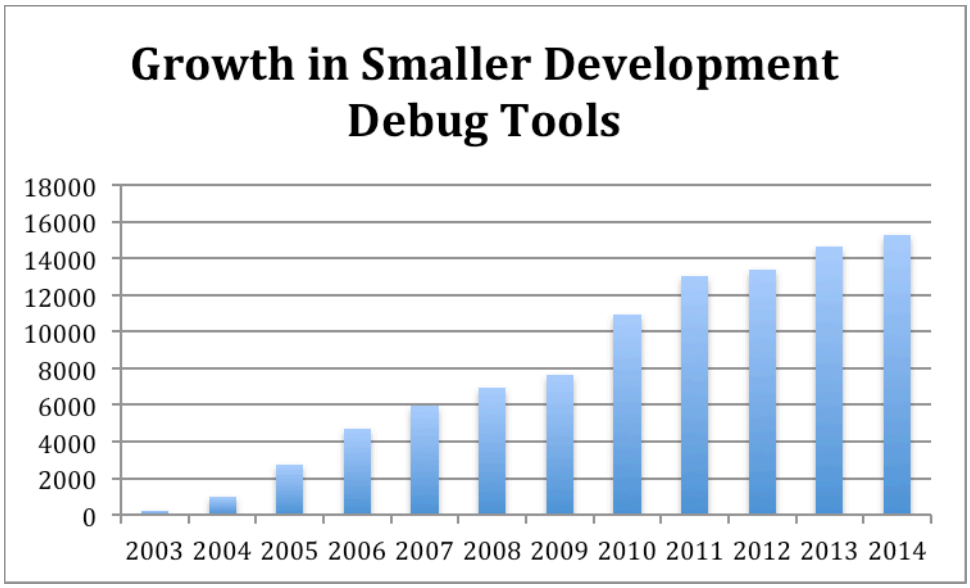


Figure 2 – Growth in Host Adapter/Protocol Analyzers Sales

The data represented above demonstrates a >40% CAGR (compound annual growth rate) for over a decade. While not typical for the overall test and measurement industry, this level of performance can be attributed to several factors:

- High portability enables both field and bench level debugging
- Faster set up times versus large test and measurement equipment drives developer productivity
- Markedly reduced time to productivity with simpler, less generic tools
- Real-time capabilities including historical capture of data
- Interoperable with other technologies via external triggers
- OS agnostic – Tools that work with any analysis system, e.g. Windows, Linux, Mac OS-X
- Lower price point accelerates purchasing decision and debugging process
- Flexibility, e.g. many capabilities within a single tool provides value
- Value proposition superior to traditional competitive technology

The combination of price point, versatility, flexibility and ease of use for these tools drives demand well beyond industry average. Beyond those high level factors, it is important to understand which specific features accelerate debugging and development?

Real time protocol analysis –
It's not sufficient for today's debugging needs to just "sniff" the protocol layer. Faster debugging is accomplished with tools that perform and provide results in real-time. Real-time sniffing facilitates iterative tests, a staple in most debugging processes. Coupling real-time sniffing with the ability to send or receive

messages on the bus enables developers to quickly find issues and test solutions.

Availability of robust APIs and GUIs to facilitate development –

- Development engineers debugging a system often rely upon the tool vendors' user interface to provide a quick and easy way to visualize issues and understand the root cause of problems.
- QC engineers automating a test process rely upon powerful and robust APIs to incorporate these powerful tools into their qualification and manufacturing processes.
- Field service engineers often rely upon a combination of manufacturer GUIs and APIs to rapidly isolate and debug in field issues.

The combination of low price point, portability, and multiple modes of interaction across diverse platforms enables firms to achieve great benefit by having all engineering functions, from project inception through post sales field service relying upon a consistent toolset. It's easier to diagnose, repair, and escalate if necessary leveraging a single consistent tool base.

Additionally, the latest generation of tools takes this concept to the next level, offering the download of additional features to enable the upgrade and customization of tools on a common platform – each role sees exactly what it needs to see exactly when it needs to see it, without ever waiting physical delivery of new tools. Enterprise administration capabilities further streamline cradle to grave support of embedded designs in a cost effective manner.

Enhanced automation –
Networked deployment of these tools is a powerful addition to the conventional individual tool approach. Now the same tools used to prototype and debug solutions can be easily be used for larger scale production line programming, offering best in class speed coupled with the simplicity and convenience of smaller tools. Eliminating reliance on large, monolithic test and program stations enables nimble production lines which are easy to keep current, offering optimal reliability and performance.

Storage (collecting or creating data)

If we look at the plethora of IT-embedded connected devices we can see an economic value that storage provides. There are numerous applications for which batch storage and periodic data transmissions make better sense than continuous monitoring. Also the ability to store historic levels of activity – on-site – can also make sense – as long as it doesn't contribute to an unreasonable power drain and as long as it is reliable and secure over a long period of time.

For example, the monitoring of road conditions, home, office or industrial based electrical usage, and remote scientific monitoring, among many similar applications, does not require constant realtime data transmission but can be effectively managed by periodic data dumps.

Medical patient monitoring can best be served by simple devices that monitor ongoing physiological data such that the period of minutes before and after a cardiac event, for example, would allow physicians to observe physiological data proceeding and following an acute event.

Holter monitors can store cardiac and other physiologic data and then do a data dump for automatic analysis. Radiologic images can be stored for comparison on request from central and physician hand held devices.

Key to the ability of IoT devices to handle such long term and highly reliable data storage requirements is an understanding of the current technologies available to embedded developers and a source of where to go to begin one's exploration and understanding of the issues surrounding IoT storage considerations.

EMF has been in touch with Roy Sherrill, CEO and founder of Datalight, Inc regarding the historic use of and the limitations of current storage devices that are appropriate for today's IoT use.

Solid State Media Realities

According to Sherrill, the past 15 years have seen exponential growth in the solid state media arena. Thumb drives replaced floppy disks almost overnight. ZIP drives became obsolete shortly thereafter. The mass storage in the iPod made CD players (and CDs) obsolete. Mobile phone cameras with digital storage have made the SLR camera a specialty item relegated to use by professional photographers.

Without question, solid state media is superior to rotating, mechanical media in many, if not most, applications. However, the technological advancement of solid state media has some characteristics which tend to be the opposite of what we

expect from Moore's Law. As a result, solid state media has characteristics which make it less than optimal for some kinds of applications.

For example, it is well known that flash wears out over time. With the year-over-year technological improvements we are used to seeing, one might assume that flash endurance has improved over the years; however, the exact opposite is true. Traditional single-level cell (SLC) NAND from the ancient days (8 years ago) was rated to be good for 100,000 write/erase cycles. Today some SLC NAND has a life of 50,000 write/erase cycles. The difference is more dramatic for multi-level cell (MLC) NAND. Whereas the life of older MLC NAND was typically 10,000 write/erase cycles, it is now down to 3,000 for today's 2-bit/cell MLC, and as low as 500 cycles for current, small lithography, 3-bit/cell MLC.
Not only does newer flash wear out more quickly, but it oftentimes does not perform as well either. Part of this is due to the stronger Error Detection & Correction (EDC) requirements of newer flash devices.

A short lifespan is not typically a problem for many consumer devices. The MLC NAND inside a mobile phone is likely to last sufficiently longer than the mobile phone itself will since typically the phone has an original owner lifespan of two years at most. The use of solid-state media in industrial devices is quite a different story, however. It is not uncommon for these types of devices to have an expected lifespan of 10 to 15 years or more. Therefore, the endurance of the solid-state media in the devices is critical.

Exacerbating the already tenuous situation of the ever decreasing media lifespan is the issue of write-amplification, whereby writing a relatively small amount of data to the solid-state media may use up a disproportionate amount of the media's endurance. All flash memory must be erased before it is written, and modern flash may never be overwritten. While it can be written in relatively small pages, typically from 512 bytes to 4KB in size, it must be erased in large chunks – referred to as erase blocks. Erase blocks range in size from 16KB to 512KB for most types of raw NAND. Most SD and eMMC media use a massively parallel array of NAND chips, and have an effective erase block size of 2MB to 8MB. Flash lifespan is measured in write/erase cycles; however it is important to note that it is really the erase portion which counts against flash life. For example, you could write absolutely nothing to the media, and erase the same block repeatedly until it was officially worn out and would no longer process an erase request. Regardless of whether there was large or small amount of data written to that block, it was the number of erases that mattered.

Therefore, if you have a 16GB flash device, which has a rated lifetime of 10,000 write/erase cycles, you could assume that in a perfect scenario, 160,000 GB of data could be written to the device before it failed (on the average). In that perfect scenario, supposing the device has 8MB erase blocks, each erase block would have to be written with exactly 8MB of client data before it is erased. If any erase block must be erased **before** the full 8MB of client data has been written to

it, then the life of the media has been artificially shortened, resulting in a characteristic referred to as write amplification. The reality is that the perfect scenario almost never happens, and therefore, some level of write amplification always occurs – the question is, “How much?”

The reality is that there are a wide variety of flash memory management schemes, from dirt-simple to ultra-complex, and many variations in-between.

With this as an introduction to developers that must include storage into their embedded IoT designs – and to the OEMs that must guarantee long term viability – the technical aspects of this subject is beyond the technical ability of the author.

Roy Sherrill, working with EMF, has agreed to answer questions from developers and managers. Although Datalight possesses technologies that overcome the limitations cited above, EMF suggests to the reader that they conduct a detailed analysis of the storage issues regarding their designs by gathering information in as many qualified places that they can find.

EMF suggests that developers and managers contact and listen to Roy as a starting point of your investigation.

Do this ***before*** you stipulate your requirements as making late term decisions are often rushed and lead to down-stream problems.

Embedded Security Overview

EMF put together this embedded security overview 5 years ago when the notion of embedded security was hotly debated and mostly overlooked. To many senior and other experienced development engineers, implementation of security protocols into embedded systems was new and confusing. Since then embedded security has been taking on greater importance – particularly as we look to IoT applications wherein a great deal of proprietary and personal data may be involved.

Although some changes have appeared in the embedded industry, EMF is repeating its previous presentation herein as an overview of security issues that are becoming essential issues for embedded developers to understand.

Embedded systems are responsible for the availability and functionality of many critical systems, from factory automation to gas pipeline monitors to networking equipment. Unfortunately, the critical importance of embedded systems is seldom matched with a strong, comprehensive security infrastructure. Some of the critical security issues presented by modern embedded systems are:

- Diverse network-connected embedded systems use combinations of custom and COTS software, the details of which are typically known only to the vendor of each embedded device, making vulnerability assessment, risk analysis, and patch management difficult
- Many embedded protocol implementations derive from older versions of open source software like OpenSSL and the BSD TCP/IP stack, resulting in vulnerabilities to known attacks, which have since been patched in the main software distributions
- Many other protocol implementations are built entirely from scratch, and have not benefited from years of public analysis and repeated attack, resulting in unproven protocol implementations that may be vulnerable to attack
- Even when vulnerabilities are identified, patches must be developed for each device or device family by the vendor, requiring tight collaboration between embedded software developers and the OEM's building devices based on the developers' software
- Deployment of software patches is even more difficult, expensive, and time- consuming than the most elaborate mobile/remote patch management systems for PCs and PDAs, making the total cost of a vulnerability in an embedded system much higher, and the motivation to patch that vulnerability much lower
- Most network-aware embedded devices lack sufficient management and auditing functionality, making centralized configuration and monitoring difficult and costly, and severely limiting the data available for attack pattern detection and after- attack forensic analysis
- Embedded systems are not always considered an IT responsibility, and thus often fall outside IT control, resulting in lax policy enforcement,

minimal configuration management and auditing, distorted risk analyses, and little or no integration with enterprise security tools

Remediation of these issues will require a concerted effort on the part of commercial and custom embedded software developers, OEM's building embedded systems, vendors selling them, and customers purchasing and implementing products based on network-aware embedded software. Until information security becomes a strategic technology for embedded systems developers, their products will continue to be characterized by complacency and vulnerability.

Encryption (or lack of)

Protection of the confidentiality and integrity of sensitive information is a critical foundational component of any secure system. Typically, this protection is implemented at least in part through the use of encryption algorithms, like DES, AES, RSA, or countless others.

Unfortunately, many networked embedded systems lack robust encryption to protect sensitive information. This may be due to resource limitations (strong encryption requires substantial processing, memory, and power), cost restrictions, design limitations, or possibly the extension of an internal, legacy, hard-wired system onto an open network such as Ethernet or IP, without considering the associated security implications.

Regardless of the reason, the potentially disastrous results are the same. Intruders or malicious insiders can read, intercept, modify, or remove communications at will. If proprietary wireless RF links are involved, the danger is further amplified, as anyone with suitable equipment can attack the system, potentially from a substantial distance given a high-gain antenna.

In many cases, damage resulting from eavesdropping on sensitive information pales in comparison to damage resulting from forged or modified communication. Consider a gas pipeline monitoring system, with wireless RF links between sensors nodes along a gas pipeline, where each sensor monitors and reports on line pressure, temperature, purity, and other critical data. If the system lacks strong encryption, an attacker could easily damage or destroy the sensors at a vulnerable point on the pipeline, then substitute his own device which generates false sensor data, while the attacker damages the pipeline. Alternatively, the attacker could generate false readings indicative of a leak or fire, diverting maintenance and response personnel from the intended point of attack.

Clearly, insufficient cryptographic protection can lead to substantial compromises, many of which are not immediately obvious at system design time. A prudent embedded system designer must consider the implications of intercepted, deleted,

modified, and forged information from all components of a networked system, and take steps to provide encryption to protect against such attacks.

Lack of Certified Encryption

Even if a system employs strong encryption to protect its security, in many cases that is not sufficient. In many markets or domains, some form of official certification must be obtained for a product or system before it can be used. A familiar example of this outside of the security field is the DO-178B certification required for embedded systems in safety-critical applications such as avionics.

In the realm of security certifications, one of the most important is the Federal Information Processing Standard (FIPS) number 140, revision 2--or FIPS 140-2 for short. FIPS 140-2 (now FIPS 140-3) is a standard and certification process for encryption which is mandatory for all information systems used by the Federal government to process “sensitive” information.

The Federal Information Processing Standards (FIPS): Why embedded vendors, OEMs and developers need to incorporate FIPS 140-2/3

Although the FIPS standards are surprisingly unfamiliar to embedded vendors, developers and OEMs, they play a crucial role in the future of embedded commerce as the new world of connected devices and network security unfolds.

The Federal Information Processing Standards are a suite of information security guidelines promulgated by the National Institute of Standards and Technology (NIST) on behalf of the United States Government. Each of the FIPS addresses a particular information processing topic, ranging from the specification of an encryption algorithm to recommended utilization of a particular security standard. Though there are dozens of FIPS, the most important FIPS, and the subject of this section, is FIPS 140-2/3.

The major fact that embedded vendors, OEMs and developers alike must understand regards FIPS 140-2/3 certification: if you offer products which perform any kind of encryption (such as IP-sec, SSL, or SSH stacks), you must affirm that all such encryption is performed by a FIPS 140-validated cryptographic module, or you can kiss federal dollars goodbye.

Of particular interest to the embedded industry is the following:

- In effect, anything that is worth encrypting MUST be encrypted using software certified under FIPS 140 version 2/3
- Federal agencies may ONLY USE FIPS 140-2/3-certified encryption products to protect sensitive but unclassified information on their computer systems. This means that even a product with certification under another FIPS, such as FIPS 46-3 specifying DES, is still not

sufficiently certified as to meet Federal acquisition requirements.

- Even if your company does not sell directly to the government, FIPS 140-2/3 certification enables government contractors doing business with the Federal government to use your product as a part of their solution.
- By obtaining FIPS 140-2/3 certification—either in-house or through licensing of a third-party component—embedded products such as medical devices, monitoring systems, alarm and surveillance systems, and all other network-aware embedded solutions can immediately differentiate themselves and open themselves to the substantial Federal government market, as well as to private contractors developing Federal systems.
- FIPS 140-2/3 is likely to emerge as the standard of "reasonable compliance" for corporate fiduciary responsibility, HIPPA and GLB.
- FIPS 140-2/3 is simply a ***cryptographic module*** certification. It is not a security protocol like SSL, and thus does not address any real-world secure protocol issues. An SSL stack, for example, could easily have FIPS 140-2/3 certification; they are not competing protocols.

Improper Application of Encryption

An equally dangerous (and less obvious) hazard to the security of a system—particularly an embedded system—is the false sense of security which arises from the use of weak cryptographic techniques, or worse still, of "scrambling" or encoding techniques which claim to provide ample security while actually providing little more than notional protection.

Using Strong Encryption, Weakly

The art and science of cryptography is a complex one; so much so that PhD's in abstract mathematics and years of experience in cryptography often design encryption systems that are found to be broken upon peer review. Of the many examples of the ease with which broken encryption can be introduced into a product, consider the widely publicized cryptographic vulnerabilities in WEP, the security component of the 802.11(b) Wireless Ethernet standard developed by the IEEE. Despite extensive input from highly qualified IEEE members, the IEEE committee which developed WEP made several significant design errors in its use of otherwise strong encryption, resulting in a very weak system.

By way of analogy, consider a chain composed of several extremely strong links, and one very weak one; despite the strength of most of the chain, the single weak link compromises the security of the entire chain. The strong-as-the- weakest-link axiom is equally applicable to system security, with the notable difference that weak links are very easy to introduce by accident, and very difficult to detect.

As a result of the substantial difficulty in the design and implementation of secure protocols, the consensus amongst the security community has it that system designers are well advised to *use existing, proven security protocols*, from existing, proven vendors, rather than develop their own protocols or implementations. Unfortunately, this is not always feasible in an embedded system, with limited resources and RTOSes not supported by mainstream security products.

Key Size Disparities

Some cases of weak use of strong encryption are not a result of an incorrect application of strong encryption, but rather from using weaker and stronger encryption together.

Recall from the previous section that a system, like a chain, is as secure as its weakest link. Therefore, all links must be equally secure, or the overall security of the system is less than the security of the individual links. One concrete example of this scenario is the increasingly common use of the Advanced Encryption Standard (AES) encryption with Secure Sockets Layer (SSL) implementations:

SSL provides a virtual "tunnel" across a network, which can be used to send information protected from interception, modification, falsification, etc. This is accomplished by using two forms of encryption together: asymmetric encryption, and symmetric encryption. The asymmetric encryption is used to negotiate a key between the client and the server for use during their communication, and the symmetric encryption is used to encrypt traffic with the negotiated key. Common asymmetric algorithms include RSA and Diffie-Hellman (DH), while typical symmetric algorithms are DES, AES, and RC4.

To continue the analogy, the asymmetric encryption is one link in the security chain, and the symmetric encryption is another. If the asymmetric encryption is broken, an attacker can determine the negotiated key, and access the traffic using that key, without having to break the symmetric encryption. Similarly, if the attacker can break the symmetric encryption, he can access the traffic without breaking the asymmetric encryption. *Given this scenario, clearly the symmetric and asymmetric encryption should be of approximately equal strength.*

Unfortunately, as stronger symmetric encryption algorithms like AES have come into common use, the corresponding asymmetric encryption has not increased in strength to match. For example, a typical 1024-bit asymmetric key is about as secure as an 80-bit symmetric key, yet AES key sizes range from 128 bits to 256 bits. To provide security equivalent to AES, asymmetric key sizes would have to range between 3072 and 15,000 bits long, which is so long that typical embedded hardware would be unable to maintain reasonable levels of performance or throughput if such large keys were used.

As a result, many systems use 128- or 256-bit AES for symmetric encryption, yet rely on 1024-bit asymmetric encryption. Thus, these systems have a security level roughly equivalent to a system using 80-bit keys for symmetric encryption. While 80 bits is still a substantial level of security, it is not the 128- or 256-bit level that is likely advertised, and the systems are incurring the additional cost of 128- or 256-bit keys, without any additional security, which is wasteful of potentially limited resources.
Though SSL was used in this example due to its near-ubiquity as the protocol of secure web transactions, the key size disparity is an issue anywhere asymmetric and symmetric encryption is used together, which is rather frequently in most secure protocols.

One appealing solution to the key size disparity problem is the promising family of asymmetric encryption algorithms known as Elliptic Curve Cryptography, or ECC. ECC uses much smaller key sizes than other asymmetric encryption techniques, while providing equally strong security. Therefore, while a 128-bit symmetric key would require an RSA or DH asymmetric key of 3072 bits in order to provide equal protection, an ECC key of roughly 256 bits would provide just as much security, at less than 1/10 the memory and processing cost. The benefits are more substantial for larger key sizes: a 256-bit symmetric key should be protected by a 15,000-bit RSA or DH asymmetric key, while an equivalent ECC asymmetric key size is only 512 bits; a 30x reduction in memory and processing cost.

Though ECC is an appealing solution to the key size disparity problems, embedded implementations of ECC are difficult to come by, and most standard security protocol implementations do not support ECC. This will surely change as ECC gains in popularity, yet for the time being, embedded systems designers face few options when addressing this issue.

Using “Pretend” Encryption

In many cases, a system is not even protected by weak or misused encryption, but rather by “scrambling” techniques which claim to level of security, but in actuality provide none. The most obvious of these is the use of spread-spectrum RF modems, which often claim 10.000 or 1,000,000 channel codes, or 1,000 frequency hops per second, as though these measures provide any protection against interception. In reality, most of these systems provide little or no protection, due to the ease with which the possible permutations can be explored, and the existence of patterns in the supposed random behavior of the scrambling process.

Furthermore, even if some level of security was provided by scrambling techniques, one must consider what happens when one of the scrambling RF modems is compromised by an attacker, either by purchasing one from the OEM, or stealing one from the system under attack. If the radio worked while in use in

the system, it will not stop working simply because it is in the possession of the attacker. Systems which depend on this such scrambling for their protection are only notionally secure, and indeed have no actual protection at all.

Unproven Protocol Implementations

Once an embedded systems designer implements encryption, obtains the necessary certifications, and resolves key size disparities, some significant security issues still remain. One of the more perilous is the issue of protocol implementation errors, and their potential security implications.

Modern communications protocols, from TCP/IP to SNMP to SSL, are complex entities, with any number of special cases, ambiguous situations, and undefined behaviors. Even with standards specifications, protocol implementers often introduce subtle and sometimes insidious bugs into their implementations, which may go undetected until an unlikely sequence of commands or malformed packets is received. Even protocol validation suites cannot detect all of these subtle errors, many of which are discovered years after the initial implementations are placed into production.

The computer networking industry is well acquainted with these implementation bugs, which often result in an insecure implementation of a secure protocol; that is, the protocol may be secure in theory, but the implementation suffers from very real bugs that compromise the security of the system. A few of the many examples of such bugs are those discovered in various SNMP, TCP/IP, and SSL implementations, respectively.

SNMP

The Simple Network Management Protocol, or SNMP, is widely implemented in various network-enabled devices to enable remote monitoring and management of disparate devices across a large enterprise from a single, remote console. In February of 2002, a serious vulnerability in several implementations of the SNMP protocol was discovered, effecting over 200 vendors. In many cases, multiple SNMP implementations contained bugs copied from other SNMP implementations, and some were in service in critical telephony and electrical systems for years prior to the discovery. Since that time, vulnerable devices continue to be discovered, and additional bugs in various SNMP implementations have also been reported.

TCP/IP

Though TCP/IP has been in use on the Internet and its predecessors for over 20 years, significant bugs and vulnerabilities in TCP/IP implementations continue to be discovered. Over the years, dozens of vulnerabilities have been discovered in TCP/IP implementations ranging from the DEC VAX to Microsoft Windows to

Linux. These vulnerabilities have ranged in severity from bypassing firewalls and hijacking sessions to causing the entire system to crash by sending a malformed packet. In many cases, the bugs were not discovered until years after the vulnerable systems were deployed, making remediation substantially more difficult.

OpenSSL-derivatives

Secure Sockets Layer (SSL) was developed in the early 1990's as a standard security protocol originally for use in secure web browsing. Since then, it has been adopted for a wide range of applications where a proven, standard security protocol is required.

Many SSL implementations exist, though one of the most popular is OpenSSL, an open-source SSL library which is also used in a number of commercial SSL products. In September of 2003, several serious vulnerabilities were discovered in OpenSSL's handing of certain malformed messages, resulting in a potential vulnerability in millions of computer systems, ranging from embedded systems to desktops and servers. Though the OpenSSL team responded rapidly with patches, it is likely that many vendors whose products are based on modified versions of OpenSSL were not so rapid in their response, as they must merge the OpenSSL patches into their modified source code, then release patches of their own, which must be distributed and installed in countless embedded devices.

Inevitable Conclusion

As the three preceding examples illustrate, implementing complex communication protocols correctly is very easy to do wrong, with potentially disastrous results. Therefore, *secure systems are well advised to use not only standard, proven protocols, but proven protocol implementations as well.* Unfortunately, this is particularly difficult in the embedded systems industry, as "proven" protocol implementations have seldom benefited from the same intense scrutiny as implementations on more common PC hardware, and in many cases are slightly modified copies of older branches of open-source projects, usually sharing their bugs as well as features.

Protecting the Wrong Things

One of the most common fallacies among security solutions is the allocation of resources to protect relatively well-defended elements of a system, while leaving defenseless those very components which are the most prone to attack. This fallacy is manifested in the embedded systems community in the form of somewhat specious security arguments regarding memory protection and process schedule enhancements as a sound security measure.

A common security argument is that memory protection—the isolation of multiple processes in a system from one another—and scheduler enhancements like CPU usage quotas protects against viruses, Trojans, and attacks against components of a system. For example, an attack might compromise a web server process, but, so the argument goes, memory protection and CPU usage quotas will prevent this attack from compromising the water pressure monitoring process, or the autopilot process.

Unfortunately, while not necessarily wrong, this assertion is meaningless in the face of modern network-connected systems and the threats they face. For example, imagine a system with two processes: one logs, monitors, and reports sensor data, and another uses IP-sec to send sensor data over the network to a central monitoring system. Memory protection and CPU usage quotas ensure that a compromise in the IP-sec process will not affect the monitoring process, either by corrupting its memory or stealing CPU time.

However, neither of these are the most significant threats. More significant threats include compromise of the IP-sec protocol due to configuration or usage errors, or attacks against the IP-sec implementation resulting in compromise of the IP-sec process, which could enable an attacker to send false readings, redirect traffic, or use the system as an attack platform. Even if one sets aside the lack of FIPS 140-certified encryption, it is clear that memory protection and CPU usage quotas do not provide the security that many believe. *A comprehensive, in-depth security strategy should include memory protection, CPU usage quotas, encryption, field-proven protocol implementations, best practice configuration, and all the other issues addressed in this report.*

Lack of Management & Monitoring Abilities

Regardless of the extent of a system's protections, it must be monitored for accidental failures and deliberate attacks, and may require management of configuration parameters, performance counters, etc. Most PC's and network equipment provide management and monitoring capabilities through an SNMP implementation (though it may be buggy; see previous section on SNMP), and some sort of logging facility, either by writing to a local log file, or using a distributed logging service such as Syslog to aggregate events into a central location. The combination of these two abilities makes it substantially easier for limited IT resources to manage and monitor a large, widely distributed ecosystem of network-connected devices, in some cases even from the remote network operations center of a managed security services provider (MSSP).

Though many embedded systems used in telecom or networking equipment already provide manageability features, many other systems—in some cases, highly constrained systems—lack even basic remote event logging capabilities. As a result, the total cost of ownership of these devices is much higher due to additional management burden, and attacks on these devices are less likely to be detected due to the IT "blind spot" created by a lack of event reporting abilities.

In many cases, SNMP and Syslog implementations are available which can be incorporated into a system design to provide the necessary functionality. In many other cases, however, standard protocol implementations are not available, and depending upon system design constraints, may not even be possible. In these situations, some form of proprietary management and monitoring functionality may be called for, perhaps with a separate component providing a bridge to standard protocols.

No matter the circumstances, embedded systems stand to benefit the most from remote management and monitoring functionality, due to their wide distribution, frequent inaccessibility, and the critical nature of their functionality. Not only does remote management and monitoring functionality contribute to the security of a system, it also lowers cost of ownership for the customer, and remote diagnostics or configuration abilities may also lower maintenance costs for the manufacturer, as routine diagnostics and initial repair orders may be conducted remotely from a central location, instead of a truck roll.

Painful & Expensive Patch Management with Minimal Accountability

For all the protective measures described above, past experience has shown that no real-world software system can be guaranteed free of bugs, regardless of the extent of security precautions taken during its design and development.

Therefore, one must assume that patch management—the process of obtaining, validating, testing, and deploying patches—will be a key component of any security strategy. In the PC networking realm, myriad patch management offerings are available from software vendors, ranging from operating-system-specific patching tools to cross-platform, cross-enterprise systems. Some tools even support embedded systems to some extent, such as those found in network devices and printers.

Any network-enabled embedded system must provide some means of post-deployment firmware updates. Ideally, the update mechanism should provide the following functionality:

- Ability to determine the current firmware version remotely over the network. Without this feature, asset tracking and patch management tools will be unable to automatically detect systems with out-of-date firmware revisions
- Ability to patch firmware remotely over the network. If this feature is absent, the substantial time and cost associated with patch deployment will act as a disincentive to patching vulnerable systems, which in turn reduces system security and increases cost of ownership
- Ability to authenticate and verify patches. If remote patching is supported, it is absolutely critical that some means of authentication is provided, so that only authorized users can apply patches. Furthermore, firmware

patches should be verified as authentic prior to application, ideally by a digital signature applied by the vendor prior to the patch release. If these features are not present, attackers could easily install firmware patches which either disable the devices, or introduce backdoors for future attack

In addition to these technical issues, the designer of a system must ensure that all software components of the system are developed and maintained according to a process which facilitates rapid development and testing of patches as bugs are removed and features added. This means that not only must software vendors provide patches; they must also employ extensive, automated regression testing frameworks, to ensure that the resulting patch fixes the intended bugs, adds the intended features, and does not break existing functionality or introduce new bugs. Without this basic assurance of validity, deploying a vendor's patches becomes a gamble, wherein one must weigh the risk from use of an unpatched system against the risk that the vendor's patch will break the system, possibly so badly as to require manual reprogramming.

Unfortunately, apart from some network devices and printers, most embedded systems provide minimal patch management support, resulting in an expensive, time-consuming patching process which discourages active patching, and minimal accountability to verify those patches which have been deployed. As long as this trend continues, the effected systems will suffer from substantially reduced security.

Compounding the problem, many highly constrained systems lack the resources to fully implement a robust patch management scheme. In almost all cases, additional costs are justified to implement over-the-network reprogramming, due to the substantial reduction in total cost of ownership realized by this feature. Failing that, all network-connected systems must provide, at the very least, some means of automatically determining the firmware revision currently installed on each device, so that even if patch deployment cannot be automated, it can at least be planned, estimated, monitored, and verified.

Unfortunately, off the shelf tools are unlikely to provide this type of functionality, as it is highly system-specific. Nonetheless, some commercial software components are available which could be assembled into a patch management solution.

Though the cost of patch management functionality may be high, the cost of a lack of patch management functionality is almost always higher; anyone designing a network-connected embedded system is strongly urged to consider the logistics of patch development and deployment, from the first line of code to the last updated device.

Embedded Limitations At Odds with Security Requirements & Existing Security Standards

In almost all cases, steps taken to increase security come at some cost, either in money, time, productivity, or resources. Thus, security is always a trade-off between the cost of additional security, and the risk mitigated by that additional security. As with most trade-offs, there exists a point of diminishing returns, beyond which the cost of additional security is higher than the risk of the breach which the additional security prevents.

For example, the cost of protecting your computer network with passwords may be a productivity loss due to time spent entering passwords, as well as a financial cost as additional IT personnel are required to deal with forgotten passwords, issuing passwords to new hires, etc. However, the cost of a compromise, such as a janitor or mail clerk accessing sensitive salary data or marketing plans, and possibly modifying that data, is much higher than the cost of using passwords, therefore few computer networks remain which do not perform some sort of authentication.

By contrast, adding three-factor authentication, requiring a password, fingerprint scan, and secure token, adds substantial up-front and on-going costs, and may not be called for if the systems in question do not contain particularly sensitive or critical information. Thus, the additional security would lie beyond the point of diminishing returns for that particular system.

This trade-off of security at the expense of other resources is particularly acute in embedded systems, where the systems may be highly sensitive and very critical, and yet also lacking in resources. Nowhere is this resource limitation more critical than in encryption protocols, which require substantial processing power and memory, which may not be available to embedded systems. A brief analysis of resource requirements of three major security protocols is presented here, followed by potential alternatives.

SSL

The Secure Sockets Layer (SSL) protocol is widely used in networked systems where the secure exchange of information is required. SSL as a secure protocol is not itself flawed; indeed it has stood up to scrutiny for years, and it currently in use in millions of web server installations around the world, protecting much of the world's sensitive information as it transits the Internet.

Unfortunately, SSL suffers from four significant limitations, which may make it unsuitable to use in an embedded system:

First, the SSL protocol is very "chatty"—that is, SSL consumes a substantial amount of network bandwidth, which may be problematic for remote systems that communicate over modems or other low-bandwidth links, in particular links which

are billed by the byte (such as cellular packet data plans) or the minute (such as long-distance phone calls).

Second, SSL relies on digital certificates to authenticate both sides of the communication protocol to each other. Unfortunately, digital certificates are based upon the concept of a trust model, which must be clearly understood and implemented by the system designer incorporating SSL into a secure system. In a surprising number of cases, systems using SSL have not employed any trust model whatsoever, which leaves SSL communications vulnerable to well-known attacks.

In turn, a trust model carries with it a management overhead, as digital certificates must be generated, signed, stored, revoked, re-issued, etc, as the system evolves. Not only does this additional management burden add to system cost, it may actually be completely infeasible given system performance and reliability issues.

Third, SSL is very often used in such a way as to introduce the key size disparity problem described in the section “Key Size Disparities” presented earlier in this section. Recall that this disparity increases system resource requirements without increasing security, effectively charging something for nothing.

Finally, SSL is almost always used over TCP/IP, which in turn adds the overhead of a TCP/IP implementation to the solution. Often, a network-connected device requires IP network connectivity anyway, in which case this is not an issue. However, for a simple sensor sending telemetry data over a serial or Ethernet cable, the overhead of SSL and TCP/IP, as well as some IP-over-serial protocol like PPP, is likely unnecessary.

In addition to the limitations above, one must also consider the issue of FIPS 140 certification, as discussed in the section “Lack of Certified Encryption” above. Even if the four limitations enumerated above are acceptable, systems which are required to possess FIPS 140 certification must by extension use an SSL implementation which implements FIPS 140 certified encryption. Unfortunately, FIPS 140-certified SSL implementations are almost non-existent, high in cost, and particularly unusual in the embedded systems space. While researching this report, not one embedded FIPS 140-certified SSL toolkit could be identified.

SSH

The Secure Shell (SSH) protocol is a lightweight security protocol often used to remotely access command prompts of UNIX-like systems. However, SSH also has the ability to tunnel any TCP/IP traffic through an encrypted “pipe”, adding a degree of security to otherwise insecure protocols, without any additional programming effort. *SSH is an immensely useful protocol for its intended purpose, but it suffers from limitations when applied to embedded systems.*

First, SSH relies upon a client's use of a list of known servers and their private key "fingerprints". When a server is first contacted, its fingerprint is noted, and if the user confirms the fingerprint is valid, the fingerprint is stored and associated with that server. In future communications with that server, if the fingerprint is changed, the user is warned of a potential attack, and asked to verify the new fingerprint. This presumes some separate means of communicating the fingerprint, some reasonable expectation that the server fingerprint will never change, and that a user is available to respond to the prompts from the SSH clients. Obviously, none of these assumptions is necessarily valid in an embedded system.

Second, as with SSL, misconfigured SSH implementations may suffer from the key size disparity problem described in "Key Size Disparities" in a previous section.

Third, SSH operates over TCP/IP, therefore it requires a TCP/IP stack. As with SSL, if the system in question needs only to send sensor readings over a serial bus or a modem, the overhead of PPP, TCP/IP, and SSH is somewhat excessive, possibly to the point of being infeasible.

As with SSL, one must also consider the issue of FIPS 140 certification in an SSH implementation. In an earlier section, "Lack of Certified Encryption", the significance of FIPS 140-certified encryption to systems used by US federal government agencies was addressed; this compliance requirement generally applies to any use of encryption in a system, and therefore would include an SSH implementation. Unfortunately, FIPS 140-certified encryption is not widely offered in the embedded systems market, making FIPS 140 certified SSH implementations particularly difficult to come by. As a result, SSH may present a technically viable solution, yet be ultimately unusable due to certification requirements.

IP-Sec

IP-sec is an extension of the IPv4 and IPv6 protocols which adds security at the network layer, transparent to all network applications. Because IP-sec is a world-wide standard for VPN deployments, and adds security to new and existing applications automatically without additional programming, it is a particularly appealing solution when security must be added to a system quickly and efficiently while retaining standards compliance. Unfortunately, IP-sec also requires some trade-offs which may make it unsuitable to embedded systems applications.

First, IP-sec is a complex protocol, with a number of possible configurations ranging from a pre-shared key to authentication with digital certificates. Each configuration has security and performance tradeoffs, and some assume a digital certificate infrastructure is already in place and available. This additional complexity increases the possibility for a misconfiguration, and thus security vulnerabilities.

Second, IP-sec's complexity makes implementation errors more likely (see the section "Unproven Protocol Implementations" earlier in this section). Therefore, relatively immature IP-sec implementations from various RTOS vendors may be more likely to suffer from subtle implementation bugs which are only eradicated after years of extensive examination.

Third, misconfigured IP-sec implementations may introduce the key size disparity problem as described in the "Key Size Disparities" section of this section.

Finally, IP-sec obviously incurs additional overhead, as well as the overhead of IP, and any tunneling protocols required to extend IP to the embedded device. While many embedded systems will require this type of IP connectivity, many others require nothing more than a secure means of exchanging sensor data and small command messages, in which case the overhead of IP-sec is somewhat excessive.

Given the discussion of FIPS 140 certification issues in the previous sections on SSL and SSH, it should come as no surprise that FIPS 140 certification (or lack thereof) is a potentially significant issue for IP-sec implementations as well. Upon reviewing the section "Lack of Certified Encryption" above, one may find that FIPS 140 certification is either a requirement or a valuable differentiator. In either case, FIPS 140 certified IP-sec implementations are particularly unusual in the embedded systems space, making IP-sec a potentially infeasible solution in situations which call for FIPS 140 certification.

Common Requirements – SSL, SSH, and IP-Sec

Given the above caveats, standard security protocols clearly incur a degree of overhead, in terms of network traffic, compute cycles, and memory. In many applications, in particular those applications for which these protocols were originally intended, these additional requirements are of little or no concern. However, in extending these protocols to embedded systems constrained for reasons of cost, power, size, or weight, the additional overhead incurred may break the system resource budget, in some cases by a significant margin.

For anecdotal evidence of this, consider the use of SSL in web applications: SSL implementations are particularly resource-hungry; in fact, almost all high-volume web sites which use SSL to secure traffic employ hardware SSL accelerators, which add additional cost and complexity to a solution. While an external and high-cost hardware component may be acceptable in an e-commerce site, it may not be suitable to an embedded deployment.

While it is difficult to estimate the exact impact a particular security protocol will have on a system, or to estimate its resource requirements, it is not unreasonable to expect a 5- to ten-fold increase in resource requirements vs. a system with no security functionality whatsoever. For systems with very low resource requirements, such as remote sensors, this factor would likely be much larger.

When SSL, SSH, and IP-Sec Are Overkill

From the above discussion, it is clear that, though SSL, SSH, and IP-sec are all robust, proven security protocols, they nonetheless are ill-suited to many of the communications problems which are common in embedded systems. In many cases, a remote sensor or other small system needs only to send a few tens or hundreds of bytes over an IP network, a serial bus, or a dial-up modem link, and does not require the additional functionality of the popular security protocols. Typically, the additional resource requirements of the popular security protocols also exceed the available resources of the small embedded system.

In these circumstances, a lightweight, simple, yet secure messaging protocol is called for, which can operate over potentially unreliable serial and dialup links, as well as best-effort packet networks like IP. Overhead must be minimal, with few buffering requirements and highly tuned encryption, to ensure maximal utilization of limited resources. Bytes transmitted must be minimized at all costs, as network communication is often the most power-hungry of all operations.

Unfortunately, no standard protocol currently exists which satisfies this need. At least one vendor is developing an open protocol to address this requirement, and at least one international standard is in development which may partially address this issue as it relates to certain short-range wireless communication needs.

Apart from them, current state-of- the-art secure networking technology does not yet have a definitive answer for the designer of highly constrained, networked embedded systems, though this situation is likely to change soon. Until then, security-conscious systems designers must make due with the overhead of existing standard security protocols, or risk suffering embarrassing, costly, and potentially dangerous security breaches.

Lack of Vendor- or Industry-Originated Best Practices

In the computer networking industry, vendors and/or industry organizations define “best practices” for securing specific products and systems, usually based on extensive experience and advice from the vendors and major users of each product. Therefore, even somewhat inexperienced systems administrators can reasonably expect to attain some level of security simply by following the relevant best practices, which typically dictate things like configuration settings, patches, etc.

Insecure by Default

One of the most basic principles among the mainstream computer security industry regarding system security is that a new system should ship with default settings which are “secure”, by some reasonable definition. This takes the form of many configuration settings, such as disabling unnecessary services, using

random passwords for superuser accounts or limiting initial superuser access to the local console, enabling encryption, etc.

Unfortunately, this principle has not yet fully permeated all PC and server software, let alone embedded systems. Designers of embedded systems continue to select insecure configurations, such as blank or well-known default passwords, security disabled, etc. This deficiency, combined with a lack of best practices which might identify the default insecurities, ensures that stock embedded systems will very often remain vulnerable to basic attacks, very likely without the knowledge of any IT personnel whatsoever.

Ambiguous Responsibilities

When a laptop or a server is brought into an enterprise, it is typically clear to all concerned that responsibility for that laptop or server lies with some information technology (IT) unit of the organization. IT units are typically equipped to ensure the security, usability, and availability of computing resources within an organization.

Unfortunately, IT is often unaware of, and unprepared to deal with, various network-connected embedded systems, from fire and alarm systems to proximity card readers to remote sensors. To the extent that IT personnel are involved with the various embedded systems which make up the network ecosystem, they are often not equipped to deal with the security implications of those systems, in large part due to a lack of understanding and best practices among rank-and-file IT staff.

As long as IT staff are not responsible for the security of embedded systems, or are not given the power and authority to fulfill that responsibility, holes will likely be introduced into the network where insecure embedded systems connect to an otherwise secured network.

Unfortunately, improving this situation will require collaboration and organization changes, particularly as it relates to IT and its interactions with other organizational units. Though this process will almost surely be slow, vendors and designers of embedded systems can take steps to ease the process by providing security best practices and guidance to IT staff, both before, during, and after initial deployment.

Large Data Solutions: How the Cloud and Watson are making data crunching easier and more cost effective

Follow the Money

We are reluctant to make unwarranted assumptions and predictions. However, after a decade of extensive surveys to determine what embedded developers have used and are currently using, we are aware of “ready for prime time” technologies that can be used to enhance the medical device marketplace. What is needed then is to look at the financial incentives needed to grow the marketplace and to see who makes money and how customers can benefit.

The emergence of the IoT is creating and attempting to assimilate an enormous amount of data. IBM forecasts that by 2020 we will be generating (from embedded devices) some 3 million petabytes of data. The key to success in the IoT is to be able to monetize the inherent value of the data acquired from all sources.

The embedded world should take a lesson from financial institutions. “Data is worthless if you don’t know how to use it to make money,” said Laura Martin, an analyst with Needham & Co. Far too many “IoT evangelists” fail to recognize the reality of this insight as they count off the many miracles available to a “willing” consumer .

For example, the only way to increase the market for medical devices by an order of magnitude is to expand the use of medical devices and medical data to the walking well. Currently, medical devices are only used (and reused) by patients that are sick.

What if, for example, the populace that was well was willing to have their vital signs, medicines and disease, surgical outcomes and ongoing status of disabilities monitored (IoT) and analyzed across millions of people? Who would benefit?

Insurance companies could determine, over millions of patient data (collectively, not individually) which drugs, procedures and guidelines are best established that would provide the best patient care at the lowest cost. Participants could receive lower cost care (and lower insurance premiums) and be provided with iPhone like devices that could not only access data but provide free cell phone use.

Currently, supermarket operator Kroger Co. records what customers buy at its more than 2,600 stores and also tracks the purchasing history of its roughly 55 million loyalty-card members. Kroger sells this information to the vendors that sell their products in Kroger stores. Unconfirmed estimates put the revenue Kroger generates at over $100 million.

This analogy can be extended to many industries such as telecom, automotive, energy and agriculture, among others.

In order to make the IoT happen, we will need Big Data providers to compete for the accrued value of extracted data. Such providers as Google, Amazon, Yahoo and IBM have the capability to provide for such.

Clearly, the best place to store this huge amount of information is in the Cloud and the best provider will possess and be able to deliver the best analytics.

Although such large and capable companies as Yahoo, Amazon and Google have the experience and technology, at this time one company has the leading edge in integrating embedded devices with IT and with performance analytics.

IBM not only has the Cloud technology already integrated into providing business analytics, but it already deploying (for use) their amazing analytic search and data crunching engine called Watson. Yep, this is the same chess-playing data engine that beat the world champion grandmasters, but it is more than a chess modeling machine.

Watson has been designed to include such artificial intelligence capabilities that can provide intuitive insights (that is outcomes that can't be anticipated by its programmers) and often counter intuitive outcomes that otherwise might have been overlooked.

In EMF's opinion, success in the IoT will be largely predicated on its ability is to not only monitor ongoing activities but to provide insights to the vast amount of collected data across the entire IoT industry. Such capabilities will provide a decisive increase in the consumption and use of embedded devices – and positively impact all of the segments discussed in this paper.

Look to IBM and its embedded and business oriented group IBM Rational to provide the glue that will drive the IoT.

Even Google, Amazon and Yahoo will benefit from Watson.

About the authors:

Jerry Krasner, Ph.D., MBA is Vice President of Embedded Market Forecasters and its parent company, American Technology International. A recognized authority with over 30 years of embedded industry experience, Dr. Krasner was formerly Chairman of Biomedical Engineering at Boston University, and Chairman of Electrical and Computer Engineering at Wentworth Institute of Technology and Bunker Hill Community College. In addition to his academic appointments, Dr. Krasner served as President of Biocybernetics, Inc. and CLINCO, Inc., Executive Vice President of Plasmedics, Inc. and Clinical Development Corporation, and Director of Medical Sciences for the Carnegie-Mellon Institute of Research. Earlier, he was Senior Engineer at the MIT Instrumentation Laboratory. Dr. Krasner earned BSEE and MSEE degrees from Washington University, a Ph.D. in Medical Physiology / Biophysics from Boston University and an MBA from Nichols College. He is a visiting professor at the Universidad de Las Palmas (Spain) where he was recognized for his work in neurosciences and computer technology.

Dolores A. Krasner received BS and MEd degrees from Fitchburg State College and Framingham State College with specializations in special education, literacy and educational research. While a full time teacher, she continued her studies with 60 credits beyond her MEd degree in educational research and best practices. Ms. Krasner retired from her full-time educational roles in 2007. Ms. Krasner is currently Senior Editor and Researcher for Embedded Market Forecasters.