

Triple Ring Technologies Whitepaper:

An Embedded Machine Control Protocol

Augustus Lowell

Chief Systems Architect

Ever-expanding computing power and increasingly complex systems are at the heart of many new medical technologies. The push toward complexity is accelerating as do our expectations about what ought to be possible. And as those expectations about what is possible expand so, too, do our expectations about the rate at which those possibilities should be realized. Product lifecycles have compressed even as products themselves become more complex. This is the challenge of modern medical product development – or, really, of development in any technology-driven industry: how do we manage both the increasing complexity of our distributed processing systems and the imperative to build them more quickly and less expensively? The answer, in part, is to have common building blocks for constructing these distributed elements and common tools for managing the interactions among them – building blocks to increase the modularity of system components and tools to encapsulate some of the complexity into manageable and validated forms. This paper describes the approach taken by Triple Ring Technologies to define and implement just such a system.

Copyright Triple Ring Technologies 2010

An Embedded Machine Control Protocol

Augustus Lowell

Chief Systems Architect

INTRODUCTION

Modern medicine is a miracle of technology. Diagnostic tests can identify the presence of complex biological markers with remarkable sensitivity. Exotic sensing modalities and sophisticated mathematical techniques for separating information from noise allow us to view both the function and structure of organs and disease within the body and with continuously decreasing levels of collateral harm. Exponentially expanding knowledge of the genetic codes and protein interactions and cellular functions that contribute to or inhibit disease – propelled by a corresponding exponential expansion in the technological tools available to observe, engineer, and synthesize the chemical building blocks that underlie those processes – open up vast new possibilities for tailored therapies. And at the root of it all, the cornerstone of that vast technological advancement is the availability of ever expanding computing power.

Data processing that once required rooms full of equipment at a cost millions of dollars can now be done in a single rack, or on a desktop, and may cost mere thousands or even hundreds of dollars. Quantities of data that once would have been beyond the imagination are now transferred, processed, and stored as a matter of routine. Where, once, the management of data was centralized it is now distributed, not only outward into a few peripheral processors but even farther, outward into sensors and actuators and user interfaces and communication channels themselves, and at any and all points in between.

Systems have become complex, sometimes breathtakingly so, and the push toward complexity is accelerating as our expectations about what ought to be possible expand ahead of the technologies themselves. And as those expectations about what is possible expand so, too, do our expectations about the rate at which those possibilities should be realized.

Product lifecycles have compressed even as products themselves become more complex.

This is the challenge of modern medical product development – or, really, of development in any technology-driven industry: how do we manage both the increasing complexity of our distributed processing systems and the imperative to build them more quickly and less expensively?

The answer, in part, is to have common building blocks for constructing these distributed elements and common tools for managing the interactions among them – building blocks to increase the modularity of system components and tools to encapsulate some of the complexity into manageable and validated forms.

A fundamental concern in any distributed processing systems is synchronization of the various processing components, and so one such building block and tool would be a common method for managing command and control communications between them, a common “language” for allowing the various modules to “talk” to each other. This would, of course, increase modularity and allow for relatively easy prototyping to the extent that any component which spoke this “language” could be connected to any other easily and with little effort expended in defining and implementing basic component interactions. But, if properly designed, such a building block would also mitigate complexity by managing some of the basic system configuration and monitoring tasks, allowing the system developer to focus more effort on the unique aspects of the system that create value and less on the basic infrastructure.

REQUIREMENTS OF A COMMON PROTOCOL

Several protocols exist throughout industry for managing some of the basic infrastructure of module interactions. The *Simple Network Management Protocol (SNMP)*, for instance, is used in network management systems to distribute basic configuration information and to alert remote servers of events that require attention. The *Intelligent Platform Management Interface (IPMI)* (and its cousin, the *Remote Management Control Protocol (RMCP)*, which is essentially an encapsulation of IPMI within UDP packets) is widely used in the telecomm and network server world to remotely identify and configure components on a dynamically-changing network, and fully incorporates SNMP along with its higher-level

functions. MODBUS is a *de facto* industry standard for low-level communications among small electronic devices, but has evolved over time such that its packet structure is different on different types of physical channels. What these all have in common is a focus on low-level infrastructure functions – for instance, monitoring of individual sensors and reporting of alarms – with very limited support for high-level logical management of complex components.

Support for such low-level module and component management is important for developing distributed processing systems, especially for medical and other instrumentation systems for which many of the individual components are likely to implement hardware-oriented functions like sensor readout. But a generalized protocol for application-level command and control of complex distributed systems should provide support for complex “black box” module interactions along with those low-level configuration and monitoring functions. In addition, it is unlikely – the industry’s experience with trying to forge consensus on the *Medical Information Bus* is a telling example – that a single physical channel (say Ethernet) will be acceptable as a communication standard across all systems and applications, from individual sensor elements to million-dollar imaging platforms. In the real world multiple physical channels, from RS-232 and RS-485 to USB to CAN Bus to Ethernet (and a range of others) will be used in various systems and even within individual systems. Hence, a generalized protocol should support operation on a wide variety of physical channels, preferably utilizing some common packet structure to minimize the need for a similarly wide variety of formatters and parsers.

In particular, we would like such a protocol to provide for:

- A full range of functional support, including bottom-level component- and module-oriented control/status queries, high-level system- and application-oriented command/control management, and movement of large configuration and operational data sets.
- Support for networks of interconnected components – not merely point-to-point communications.
- Channel-independent messaging structures to minimize the complexity of support for modules on different kinds of communication links.

- Discretized (“packetized”) message structure to support discrete command/control interactions.
- Multi-packet messages to support transfer of large data sets.
- Application-level control of packet sizes to allow for optimization in overlapping data read and transfer operations.
- Dynamic identification of components as they are connected to the network.
- Hardware-oriented transactions for low-level device management.
- Application-oriented transactions for high-level system management.

EMBEDDED MACHINE CONTROL PROTOCOL

Rather than assembling several existing protocols and *ad hoc* extensions, with limited functionalities and overlapping capabilities, into a quasi-standard customized implementation, building a new protocol standard from scratch allows for an integrated design. The *Embedded Machine Control Protocol (EMCP)* is a packet-oriented protocol designed specifically for management of distributed processing elements within an embedded environment. It was developed over the course of several years for use in a large-scale x-ray system incorporating several dozen processors distributed over multiple equipment racks in several rooms and interconnected both by modern networked communication channels and legacy point-to-point links. That initial implementation was expanded and standardized within Triple Ring Technologies as the standard protocol for use in medium- and large-system development.

The EMCP is an application-layer protocol designed to operate on top of a lower-level packet protocol like UDP and/or IP (in a networked environment) or like PPP or HDLC (in a point-to-point environment). Because it is not assumed that the underlying protocol will guarantee message delivery, the EMCP provides mechanisms at the application layer to verify message delivery to the actual end-point – that is, to the application rather than

merely to the communication protocol receiver. The EMCP provides the following features:

- *Standardized mechanisms for module identification:* Standard formats for module and class identifiers, which include manufacturer and functionality codes, are incorporated into a general “Identify” query that may be requested at any time. In addition, each module should announce its presence to the network at the time it comes online; that announcement may be in the form of a unicast, multicast, or broadcast depending on the capabilities of the underlying transport protocol.
- *Standardized mechanism for module control/status management:* Standard formats for common module control and status functions – including a general “enable/disable” module command – are incorporated into general control and status messages. Provisions are made within the messaging protocol for module-specific command/status extensions, but every module should support the minimum set.
- *Standard mechanism for identifying individual messages and associating responses with queries:* Individual messages are tagged with unique message identifiers and coded as commands, queries or responses; queries require a response, and the responses are tagged with the same identifier as the query that instigates them, allowing the response to be associated with the original query at the receiver.
- *Application-level control of packet fragmentation:* Allows the communication channel to be tuned for optimum latency when large data sets are requested from relatively slow hardware buffers. The application may choose a packet size which matches the internal data readout time to the link transmission time in order to optimize the overlap between data readout and transmission.
- *Standard I/O-oriented packet extensions:* Supports low-level hardware accesses and/or creation of virtual I/O spaces directly addressable through the protocol. This is particularly useful for board bring-up, diagnostics, and interfacing to minimally “smart” modules which implement their communication end-points (for instance) within an FPGA.

- *Hooks in the header for custom extensions:*
Flags are provided in the header for signaling extended functions, similar to the I/O functions. This allows the protocol to be customized in such a way that parsing of extended functions could be handled directly as a parser extension rather than at the application level; a flexible implementation would allow installation of these extensions as a run-time or compile-time option.
- *Standardized command/query language:* Full set of commonly-used parameterizable message packets like “Get Data”, “Set Data”, “Send Command”, and so on. Specific commands or data sets are specified by identifiers within a packet fields, and support for these standard packets is provided in the nominal EMCP implementation in such a way that it allows compile-time or run-time installation of handlers for specific identifiers. This simplifies development of applications that make use of these common language templates.

Header Format

The basic header for an EMCP packet is shown in Figure 1; a non-fragmented, non-I/O packet comprises 24 bytes arranged as 4-byte values. The use of 4-byte values for the basic header structure not only provides a high degree of flexibility in use of the fields themselves, but also makes the header maximally compatible with communication protocols and processors that operate most efficiently with double-word data.

Both destination and source identifiers are provided in each message to allow for filtering of message responses based on where it came from and to whom it was addressed. In particular, this allows individual modules to tailor their responses to broadcast messages used for system-level management functions, but it also allows for more esoteric application-specific customizations.

**<Dest. ID><Src. Class ID><Sequence #><Error Code><OpCode><Data Length>
[<Chunk Length><Chunk #><Chunk Count>]
[<I/O Port Width><I/O Port Offset><I/O Transfer Count>]
<Data>**

Field	Size (bytes)	Description
Dest. ID	4	Receiver's ID for broadcast support
Src. ID	4	Sender's ID for request validation
Sequence #	4	Transmitter-assigned packet sequence # for use in matching responses to requests
Error Code	4	Error code from responder
OpCode	4	Type of transfer operation requested
Data Length	4	Length of the Data field in bytes
Chunk Length	4	Size of a single multi-packet data "chunk"
Chunk #	4	Index of the current multi-packet data "chunk"
Chunk Count	4	The number of data "chunks" into which a multi-packet transfer was divided
I/O Port Width	4	Width of I/O port
I/O Port Offset	4	Offset of I/O port
I/O Transfer Count	4	Number of I/O transfers
Data	Variable	Packet payload, varies with packet type

Figure 1: EMCP Packet Structure

The sequence ID provides a unique identifier for each message to allow association of responses to queries; it is also used for

fragmented (“Chunked”, to differentiate it from any fragmentation done by the lower-level protocols) messages to assist in re-assembling the fragments. The error code is provided to allow responses to indicate error conditions even if they return the requested data. The OpCode and data length encode the specific message function, including identifying the amount of data that follows the header.

As shown, the I/O and “Chunk” management fields are optional, and are not included in a simple non-I/O, non-fragmented packet. This allows the protocol to minimize the header size for simple command/control/status operations. Presence of the I/O and “Chunk” fields is signaled by bits in the OpCode field; the OpCode field also reserves bits for use by the specific implementation to signal additional optional header fields that could be used for functional extensions.

In addition to flags indicating the presence or absence of optional header fields, the OpCode also includes flags indicating whether the packet is a command (no response required), a query (response required), or a response (only transmitted as the result of a query). Because these fields are present in the header, the parser itself can implement matching of responses to queries, and can signal an error if an application-defined timeout occurs before a response is received. Hence the query/response handshake can be used to verify message transmission all the way through to the EMCP end-point, and no support for message acknowledgement is required by the underlying protocol.

The OpCode field also contains a 20-bit function code, which is used by the parser to determine the specific operation being requested and which message handler to invoke in response to the message. Twenty bits provides for up to a million different function codes, allowing creation of very complex systems and for the organization of functional subsets within well-defined op-code ranges.

The optional I/O fields are used with “I/O” messages to create access to a virtual I/O space. This is, in effect, a secondary access device mechanism that operates in parallel with the logical “black box” transaction mechanism defined by the standard protocol. The I/O fields designate the offset, port width, and element count for the transfers; the function code in the OpCode field defines the specific type of I/O operation to be performed.

Module Identification

Source and Destination module identifiers can be one of two forms, a Module ID or a Class ID. Which is used is signaled by a Class ID flag in the MSB of the ID word.

A Module ID identifies a particular and specific type of module. The Module ID contains sub-fields to specify a *Function ID*, a *Manufacturer ID*, and a *Type ID*. The Type ID is roughly equivalent to “model number”, assigned by particular manufacturers to differentiate their different products. The Function ID is intended to allow identification of functionally-equivalent modules with “standard” APIs – say a device defined as a “generic” A/D converter using a standardized ADC API, or a device which adheres to some industry-standard API specification.

Class IDs are typically define at the application level to indicate functions relative to the system under development. For example, a designer might define a class ID indicating the system master; messages identified as coming from the “system master” could then be treated differently than messages identified as coming from (for example) a “subsystem manager”.

Standard Messages

The EMCP defines the following standard message functions, designated by the OpCode and supported by pre-defined structures in the data fields:

- *Reset*: Reset the module
- *Enable/Disable*: Globally enable/disable the module functions
- *Status*: Report module status. 8 common status flags, indicating whether the module is operating, the type of the last reset, and general fault conditions; 24 bits are reserved for use by the application to indicate application-specific status items.
- *Errors*: Reports of various link and module error conditions
- *Identify*: Report the module/class ID and a string representing a textual module name
- *I/O Configuration*: Enumerate the valid virtual I/O space configuration for I/O messages

- *Hello/Hello Ack/Goodbye*: Asynchronously announce the module on the network when it comes online or goes offline
- *I/O Read/Write Block*: Read/write a block of data through a single I/O port
- *I/O Read/Write Table*: Read/write a block of data through a group of contiguous I/O ports
- *Command*: send a command; parameterized by a command ID. This is the standard “do something” message.
- *Publish*: Share a globally-exposed data value to other modules on the network; parameterized by a data ID; allows variable data size based on data ID.
- *Set Parameter/Get Parameter*: Set or get a “parameter” – a data value identified by a parameter ID – to/from a module; allows variable data size based on parameter ID. This is the standard “Data Transfer” message.
- *Notify*: Signal an event to other modules on the network; parameterized by an event ID and accompanied by an event data packet; allows variable data size based on event ID.

The last four messages comprise the standard parameterizable message interface that can be used for common module implementations. The parser handles these messages in a general way and dispatches to application-installed handlers indexed by the ‘ID’ values provided with the messages.

We should note that the *GetParameter* and *Notify* messages can be used as direct equivalents for the sensor and event messages provided by protocols like SNMP and IPMI. Hence, although they are not specifically defined by the EMCP protocol, the EMCP protocol provides the mechanisms necessary for board-level management functions, including module identification, sensor monitoring, and event notification.

Benchmarks

The EMCP was developed to simplify the creation of large, highly-distributed processing systems; and, as is apparent from the

structure of the header, the EMCP was intended to be extremely flexible in order to be applicable to a wide range of applications. That, of course, comes with a cost: the header itself is 24 bytes long – roughly on parity with similarly complex network protocols designed for other applications – and is therefore very inefficient for the transmission of small data packets, especially through relatively low-speed links.

Although we stipulated that the EMCP should be compatible with a variety of physical channels, including RS-232, it may not be the best choice for a system that is implemented primarily using such low-speed channels. Within our own environment we have operated the protocol using an RS-232 link at 115.2 kbaud but, as might be expected, the throughput, although reasonable for large data transfers (like images), was severely diminished for small command/control packets.

For reference, Table I shows the estimated EMCP performance for various common physical channels. We assume the entire message fits within a single packet for the underlying protocol; for the point-to-point links (RS-232 and USB) we have assumed the underlying packet protocol is provided by PPP (*Point-to-Point Protocol*), while for the network channels we have assumed UDP/IP as the underlying protocol. For USB, the estimated performance is based purely on link performance, without taking into account the polled nature of the USB channel and the resultant inefficiencies sometimes introduced by the drivers within the operating system. For network channels the performance is estimated strictly from the packet encoding, without accounting for any non-packet-related overhead associated with network transmission.

As we can see, even over an RS-232 link the available performance – provided you can run the link at 115.2 kbaud – is roughly equivalent to what was traditionally available from a 9.6 kbaud link, even for very small data transfers. Greater efficiencies can be obtained by packing control and status transfers together within a single packet rather than in separate packets, although that increases the complexity of the application required to deal with the packed packets. For relatively large data transfers the large header size becomes relatively unimportant, and over modern network and USB channels the throughput is reasonable for command/control applications even for small messages.

Table 1: Estimated EMCP Performance on Various Links

Protocol	Raw Rate	Overhead Rate	EMCP rate	
			Data=4 bytes	Data=1024 bytes
RS-232 w/PPP	115.2 kbaud	92.16 kbaud	10.84 kbaud	89.54 kbaud
USB w/PPP	12 Mbps	11.62 Mbps	1.37 Mbps	11.29 Mbps
10-base-T w/UDP	10 Mbps	10 Mbps	678 kbps	9.49 Mbps
100-base-T w/UDP	100 Mbps	100 Mbps	6.78 Mbps	94.9 Mbps
1000-base-T w/UDP	1 Gbps	1 Gbps	67.8 Mbps	949 Mbps

The primary advantage of the EMCP, of course, is not performance but standardization. To the extent that all modules may be controlled, through any type of physical channel, using a common command/control protocol application development becomes much simpler.

CONCLUSION

The EMCP provides a common command and control protocol for the design of complex distributed processing systems. Use of such a common protocol facilitates modular design and reduces the complexity of system management by providing standard mechanisms for module identification and control, and common formats for command and data transfer.

REFERENCES

- 1) *IPMI, Intelligent Management Platform Interface Specification, Second Generation v2.0, Rev. 1.0, 12 Feb 2004, Intel/HP/NEC/Dell*
- 2) *MODBUS Over Serial Line Specification & Implementation Guide, v1.0, 2 Dec 2002, Modbus.org*
- 3) *RFC1098, Simple Network Management Protocols (SNMP), Apr 1989, IESG*