

Control Protocol Design

A Guide for Controllable Hardware Manufacturers

Dean Roddey
Charmed Quark Systems, Ltd.
www.charmedquark.com

Last Edited on: December 25, 2012

Table of Contents

Table of Contents2

Content Summary3

Prerequisites.....3

The Basics4

The Words Used5

 Verbosity.....5

 Word Order5

 Strict Definition.....6

 Which to Use?6

Sentence Boundaries7

 Binary Strategies7

 Header+Payload.....7

 Start/End Marker7

 Start/Length.....7

 Start/End + Header.....8

 Textual Strategies.....8

 Start/End Markers8

 End Marker.....8

 Header/Length8

 Combination Strategies.....8

 Ambiguity Issues9

 Efficiency9

Syntax..... 10

 Binary Syntax..... 10

 Textual Syntax 11

Flow Control 13

 Low Level Flow Control..... 13

 Call/Response vs. Async Notifications..... 13

Acknowledgements 16

Sanity Checks 17

 Checksums..... 17

 CRCs (cyclical redundancy checks) 17

 Encryption 17

 Magic Markers 17

Miscellaneous Concerns 18

 Encryption 18

 Login 18

General Rules..... 19

 Support Multiple Users..... 19

 Device Response Times 19

 Dead If Off..... 19

 Minimum Inter-Message Intervals 20

Sample Protocols..... 21

Content Summary

This document is for developers of hardware devices which are intended for external control by automation systems, specialized control applications, or hardware controllers, via some sort of wired or wireless connection. Any such device must expose a 'control protocol' which defines the form and meaning of commands and information passing between the two involved parties. This document describes how to create a high quality control protocol, and the considerations thereof.

The information in this document is intended to improve the quality of automation at the most fundamental level, because automation systems are no better than the extent to which they can reliably and conveniently provide control over the customer's devices, and long experience has shown that far too many devices have substandard control protocols.

This document starts out fairly simply and generically, so if you already have some experience in this area, you may wish to skip forward a bit to the more detailed technical sections.

Prerequisites

To gain a full understanding of the concepts in this document, you should have a grasp of the following topics, though any reasonably technical person will gain very useful information, interspersed between chunks of seemingly alien communications.

- *The basic concepts of software engineering, such as bytes and bytes and structures.*
- *The basic concepts of automation, which are too broad to really attempt to enumerate and explain to any great degree here.*
- *The basic concepts of data communications.*

This document will for the most part be independent of any particular communications medium. I.e. it makes little difference at the protocol level whether the device exposes itself via serial port, wired Ethernet, wireless Ethernet, USB, etc... The medium used will change the specific interfaces used to receive/send messages back and forth (often referred to as reading and writing messages), but they aren't generally important in terms of the design of the protocol itself, which is a higher level construct, much as it makes little difference if two people talk over a telephone, walkie-talkies, or a string stretched between two cans. They are all just a means to move the information from one party to another, but the information is the same. Choice of medium is a bit more of a marketing and cost decision than a technical one.

The Basics

At the most fundamental level, a control protocol defines the rules that two 'parties' use in order to have a coherent conversation. When two people meet on the street, there is an implicit control protocol between them that allows them to communicate (mostly) successfully. Sometimes they may not follow the rules as strictly as they should, as we all know too well, but mostly we all follow the protocol. The important aspects of such an exchange, as defined by the protocol, are:

- **Words Used.** If the two sides don't agree on the meaning of the individual sounds or bytes or electrical pulses they are throwing towards each other, not much useful is going to come of it, e.g. two people who don't speak the same language.
- **Sentence Boundaries.** Even if both sides understand the words being used, if both sides cannot recognize the start and end of a chunk of words that are intended to be taken as a logical unit of communications, then dialogue will be very difficult. This is of course similar to punctuation in written information, used in the delimitation of phrases, sentences, and paragraphs.
- **Syntax.** Even if both sides understand the individual words being used and how to break them out into consumable chunks, if the contents of those chunks are not arranged in some mutually understandable syntax, it will still be very difficult to communicate, e.g. two people who speak the same language very badly and put the nouns and verbs in the wrong place and whatnot.
- **Flow Control.** Some humans have problems with this one, but each side of the conversation can only absorb information (data) and react to it so quickly. So there must be some way to tell the other side, please slow down and let me get my head around what you've already told me before you tell me anything else. Note that this is different from sentence boundaries above, which is about delimiting individual groups of words intended to be consumed as a unit of communications, sentences basically. Flow control is about the rate of arrival of such sentences.
- **Acknowledgement.** In even fairly casual conversation, and certainly in important conversations, we have means of acknowledgement, which are used by one side to give a positive indication to the other side that, yes, I have heard what you said and I acknowledge that I understand it. Or in some cases a negative acknowledgement, that you were not heard or understood.
- **Sanity Checks.** We often, in our conversations will use various types of sanity checks, such as forcing the other side to regurgitate what we have said, to convince ourselves that they do understand us and/or haven't missed important information, or use various gestures, expressions, cross references, and side commentary to insure that the information is correctly understood.

All of these concepts are completely relevant to the conversations had between a device being controlled and the thing doing the controlling, which we will henceforth refer to as the device and the controller for brevity, and to avoid being too specific about the nature of the two sides of the conversation. Both could be software programs on the same network or on the internet, both could be small embedded devices, hand held devices, or any combination of these. Each of the above concepts will be examined in more detail below in separate sections, so that they are well understood before moving on to higher level considerations.

The Words Used

In this section we will explore the most fundamental aspect of communications between controller and device, that of the words used in the communications. As a practical matter, what this ultimately comes down to is whether the protocol is 'binary' or 'textual'. As a simple example it would be the difference between using a numerical 0 or 1 bit to indicate off or on vs. using text to indicate the same using perhaps the English words 'Off' and 'On' or the characters '0' or '1'. All protocols are typically one or the other, though of course even a binary protocol must occasionally or commonly transmit text information within it. The differences will become more obvious in the next section where we discuss the level of syntax.

- *Of course ALL digital communications is really binary technically. All computers understand are bits and bytes. But computers deal with text quite easily via the concept of 'text encodings' which we'll discuss below.*

Many people have trouble understanding this difference between binary and textual communications, and the important related concept of text encodings, so we'll take a short detour to explain it. In order to have computers deal with text, given that it only really understands numbers, we have to come up with some scheme to represent (to encode) text as numbers. We could say that 1 is A, 2 is B, 3 is C, and so forth, with perhaps 27 through 37 being 0 to 9, and with a couple numbers for punctuation characters, and that could very easily be used to communicate information between two computers either directly or by storing it in a file that one writes and another reads back in, and so forth. As long as both the reader and writer agree on what numbers represent what characters, digits and punctuation (in other words the text encoding used), there is no ambiguity and we have successfully used numbers to encode text.

There are many different such text encodings out there, and you may see them referred to sometimes in e-mail or HTML files. They have names like UTF-8, ISO-8859-1, Unicode, Windows 1252, ASCII, and so forth. Don't let them intimidate you since all they are is a mapping of meaningful characters to numbers so that text can be stored and manipulated in a computer. The only real concern you have is that when your computer or device or controller reads in some text, that it knows the encoding that the text was sent in so that it can correctly decode that text. In the e-mail and HTML worlds the MIME system provides this indication, and you will see something in the header of an HTML file like "text/html; charset=ISO-8859-1", which means that the contents of the file is text, formatted as HTML, and the encoding is ISO-8859-1. That tells the receiving system how to interpret the text received.

Verbosity

There are some side effects of using a textual protocol, such as the representation of numbers or Boolean (off/on or false/true) flags. Whereas we could represent a number like 127 as a single byte in a binary protocol, that can't be done in a textual protocol. Instead we have to use three bytes, which in our simple encoding above would be 28,29,34, i.e. the values that represent the characters '1', '2', and '7'. We could save one byte in these situations by using a hexadecimal (base 16) representation which would only require two digits to represent a byte, from '00' to 'FF'.

So, clearly textual protocols can be more 'verbose'. Whether that is an issue or not depends on the circumstances. In today's high powered world it's not generally an issue, particularly if the amount of information transmitted isn't usually that large and therefore the size differential isn't really substantial. It may also depend on the syntax of the text being used. XML and HTML, for instance, are fairly common and both can be quite verbose, perhaps taking up to ten or twenty or more times the number of bytes required to transmit the same information in a compact binary form, or even a more compact textual form.

If the communications medium used is very slow, verbosity can potentially become an issue, e.g. protocols used over wide area wireless networks can be performance sensitive. For local area network, USB, or serial connections, it's not generally too much of an issue.

Binary protocols are capable of being highly compact generally, so verbosity isn't too much of an issue, at least not from this word level perspective. However verbose your syntax is (covered in the next section), the use of a binary format will typically insure it can be transmitted in the least amount of bytes possible.

Word Order

In a binary protocol, we run into the issue of word order. A single byte in a modern computer will likely always be 8 bits and can represent an unsigned value from either 0 to 255, or a signed value from -128 to 127. But, if you need to represent a value of greater

range than that, you have to use more than one byte. When such values are written out to a file or transmitted over some medium to a receiver, one has to come first and the other has to come second (and so on if more than two are required.) If the two sides don't agree they will interpret the numbers very differently. The order of these bytes is referred to as the 'endianness' of the protocol, where 'big endian' means that the most important byte comes first, and 'little endian' means that the least important byte comes first. So it is very important that a binary protocol define the endianness of the data, and that each side swap the bytes of any multi-byte numerical values as required on the way in and out.

Textual protocols, though they use more bytes, don't have this issue since they just spell out the numbers with textual digits, which the receiver must then convert to a binary value for internal use. The receiver understands that a number like -14 would be transmitted as characters in the natural written order, i.e. '-', '1', and '4' (using whatever actual numbers those encode to in the chosen text encoding.) So text protocols are endian neutral.

There can also be some other issues such as the format of floating point numbers (if any are even involved in the control protocol and often they are not) or the form of negative numbers. There's no negative sign in the binary world. Both sides have to agree that a given number is either a signed or unsigned value and interpret it accordingly. Signed numbers are encoded in a specific way, and both have to understand what form that is. But it's very typical these days that any controller or computer or device is going to use the very common two's complement form for signed numbers and the IEEE format for floating point numbers. So this generally isn't a worry. The device can indicate the use these forms in the protocol, and in the rare case of a controller that doesn't use them natively, it can translate them as required.

Strict Definition

One key ingredient to a good protocol is very strict definition of every aspect of the information being passed back and forth, and the ability to add new features to the protocol without breaking existing users of the old version. The biggest problem with textual protocols is that non-readable characters can be included into the text without being visually obvious. This may or may not affect any given consumer of the data, depending on what assumptions it makes in its parsing of the text. So the tests of the manufacturer may not catch a problem that ultimately breaks numerous existing controllers after some firmware update of the device.

Binary protocols, which are generally defined by a 'structure', which is a programming term for a strict layout of values in memory and something that is supported directly in many programming languages, don't tend to have this issue. They are very strictly defined in this way and there's typically not as much opportunity for assumptions to be made by the consumers of the data.

We'll get into some specific examples in the Sentence Boundaries section below.

Which to Use?

There are good arguments for using either binary or textual form for your control protocol. The author would argue that binary protocols are typically more strictly defined and efficient. However, because of the capabilities (one could say limitations) of many modern programming languages, which do not allow for arbitrarily spelunking around in a block of memory for safety reasons, a textual format can often be more practical, since the block of memory can be just treated as a stream of text that can be parsed out sequentially using built in language capabilities.

In terms of complexity, binary protocols are typically more complex initially to get into, because a text protocol can often be easily explored using a simple text terminal program connected to the device. The text output of the device just shows up as lines of text on the terminal program. In theory a well written protocol document should make this unnecessary, but it still tends to make people comfortable, and allows for an easy visual sanity check during the development process.

Ultimately though, either will work well as long as they are of good quality. It's generally a six of these, half dozen of the other, sort of proposition. In terms of what's out in the wild, the author would say that text based protocols outnumber binary probably ten to one if not considerably more. But that almost certainly has less to do with the inherent technical superiority of a textual protocol than the ease of implementation.

Sentence Boundaries

Once you have defined the form of the words of your protocol, you need need to provide a means for two sides of the conversation to be able to recognize coherent chunks of words intended to be consumed as a unit, e.g. the sentences in the protocol language, often referred to as messages. This extraction of individual messages is a key element of both the controller and the device and must be done reliably, and both sides will typically have some low level code that handles this task, .e.g. a GetMsg() function. Higher level code will call this function and expect it to give back the next message received (or perhaps an error that it didn't receive anything or that what it got was malformed and so forth.)

Making this aspect of the protocol very unambiguous is a big part of good protocol design and it's not terribly difficult to do. It will typically be done quite differently in binary vs. textual protocols, but sometimes the same sorts of schemes are used. We'll discuss various strategies in the context of binary and textual formats.

- *Note that this is separate from sentence syntax, which we'll discuss below. Though some consumers of messages may choose to parse and validate the contents of the message as they are receiving it, often the reading in of individual messages is a separate thing from the examining and validating of the contents of the message.*

Binary Strategies

There are a number of strategies used in binary protocols. Because of the fact that binary protocols very easily allow for fixed layouts of data, what would be termed 'structures' in a programming language like C or C++, lots of binary protocols use some sort of structure to represent the data being transmitted. As long as both sides agree on that structure, it becomes fairly easy for both sides to send and receive messages, once any word order issues have been dealt with by flipping the order of bytes where needed.

Header+Payload

A very common scheme involves the definition of a 'header' that is the same for all messages, plus a message specific payload part that follows it. This header will typically at least include the type of message that the payload contains, and the number of payload bytes. This allows the receiver to generically read in and do basic validation of incoming messages. It can look at the information in the header, verify that it looks like a valid header, and then read in the number of payload bytes indicated. By having a distinct header and payload, a low level message reading function can just discard the header once it has verified that it has read a good message, leaving only the specific message part to further process.

In order to handle the problem of getting out of sync with the sender, the header will often contain various types of sanity checking information, which will be covered in a later section.

Start/End Marker

Another possible scheme, though less commonly used, is that of a start/end marker so that there is a start byte, followed by all of the bytes of the message, followed by an end byte. Getting in sync is fairly easy with such protocols since the receiver can just scan forward still it sees a start marker then read till an end marker is seen. Once the receiver has received the message, discarding the start/end markers, it should have a good message.

This scheme can be problematic in binary protocols due to possible ambiguity, which is discussed at the end of this section. Dealing with the ambiguity can offset any simplicity benefits this scheme might have for a binary protocol.

Start/Length

This is really sort of the same as Header+Payload, but it really doesn't quite qualify as a header because it's just a start byte followed by a number of bytes to read. So the reader looks for a start byte, reads the length byte that follows, then reads the indicated number of bytes.

This scheme, like the Start/End Marker scheme, suffers issues of ambiguity in binary protocols, exacerbated by the fact that if one incorrectly senses a start byte, and therefore reads a bogus following length byte, it could issue a read for the wrong amount of data and make the problem even worse.

Start/End + Header

Sometimes the start/end is combined with the Header and Payload scheme, which provides a sort of cross reference sanity check since the length byte in the header can be used to read that many bytes, the last of which should be the end byte if things are working correctly. If the start/end values are, say, 32 bit values with fairly randomly selected bit patterns, the odds of them just happening to occur in those locations are pretty small.

Textual Strategies

The textual strategies tend to be easier because of the simple fact that the textual encoding makes it fairly easy to guarantee that bytes used as start/end type markers will never be used in the actual body of the message. A binary protocol may legitimately use any of the possible 256 byte values in the body of a message. But a textual encoding generally only needs the upper and lower case characters, the digits, and maybe a few punctuation characters. That leaves plenty of the possible 256 values of a byte available for use as unique message delimiters. Some encodings have characters specifically designed for this purpose. These are typically 'non-printable' characters, i.e. they are not used to convey text content, but purely to delimit text content and be removed before display. See the discussion of ambiguity below.

Start/End Markers

This is arguably the most commonly used strategy for textual protocols. Special byte values are used to mark the start and end of a message. This makes for a very easy sync strategy as long as ambiguity isn't an issue. The receiver scans for the start marker, reads up to the next end marker, and he has a whole sentence of the protocol. The start/end markers are discarded, leaving just the message. There's little danger of getting out of sync if one side connects in the middle of the other side sending data, since the receiver always just scans for a start marker and anything it sees before that marker is discarded.

End Marker

Sometimes there is only an end marker, with the start marker being basically implicit. I.e. the next byte after you read the end marker must be the first byte of the next message. This is not quite as robust since if junk data is somehow transmitted, perhaps partially seen incoming message upon first starting up, it cannot as easily be caught and discarded, until the program has gone through the work of trying to parse the incoming message.

A common scheme here is a newline separator. Many systems have built in tools to parse out newline separated lines of incoming text, so this can make it easy to implement such schemes. But it's not a recommended one, because it doesn't make for a strongly defined format and such protocols can be easily broken by changes that were not foreseen by protocol implementers. Also many platforms don't agree on what constitutes a new line, thus obviating the advantage of having a built in mechanism for parsing new line delimited values, and inconsistent newline values can be introduced without the manufacturer noticing this in testing.

Header/Length

It is possible to do a sort of header/length type of scheme, even with a textual protocol. It's not so commonly done, because an unambiguous start/end or end marker scheme is so easy and works well, but there are devices out there that use this scheme, sometimes because they want to provide sanity check information that is separate from the main message payload, as in the binary world above.

Combination Strategies

There are some devices that might combine a binary plus text scheme. I.e. they might use a simple binary header, which includes an indicator of the number of payload bytes to read, which is in some parseable text format. The XML Gateway Server of our own CQC product uses such a system. There is a binary header, followed by an XML based payload that the receiver parses out using an XML parser. This type of scheme combines some of the efficiency and convenience of binary for the reading in of messages, with the flexibility (but also the bloat) of text for the content of the payload.

Ambiguity Issues

Of course any scheme where you use a language to describe other data expressed in the same languages raises issues of ambiguity. HTML is a case that many people may be familiar with. HTML uses text to describe aspects of other text, e.g. the color, font size, etc... This requires that there be a way to distinguish between the data and the metadata (the data about the data.) The tags are the metadata, and the text actually displayed to the user is the data. Normally this is fine, until you have to display some text that might use some of the special delimiters, or even if you need to display HTML tags as the actual text, for instance a web site that demonstrates how to write HTML and therefore must treat HTML tags as both data and metadata.

Control protocols suffer the same problem sometimes. If you have delimiter information such as start/end markers, you have to have a way for the receiver of the message to be sure that he is reading a message at the start of the message, not just picking up somewhere in the middle, and when to stop reading and process the message received.

In text protocols, this is not too hard. As mentioned above, most text encodings specifically include special characters not designed to display text, but to delimit text, and those can be used very conveniently in textual protocols. One trick commonly used in textual protocols designed around the ASCII text encoding (lots of them) is to use the STX/ETX characters (binary codes 2 and 3 in the ASCII encoding) as the start/end markers.

It is quite easy to arrange for these characters to never actually be used in the body of any legal message, so these markers are completely unambiguous. If a sender did use them, that sender would be so broken that it's not worth worrying about. Many of the extant text encodings tend to include the ASCII set as part of it, and just extend it in one way or another, so the same STX/ETX characters are likely to be available in other commonly used encodings, such as Windows 1252, UTF-8, and Unicode. So a message in this scheme might look like this, where <STX> and <ETX> are spelled out for readability, though they would actually just be single bytes encoding those characters:

```
<STX>the message content<ETX>
```

For binary protocols, the issue is a bit trickier, because any of the 0 to 255 values a byte can hold might legally be a value in the body of a message. A common scheme is to use something like the Header+Payload format discussed above, but that doesn't guarantee complete non-ambiguity unless the magic marker values in the header are guaranteed not to ever exist in the payload of a message in the same distances from each other. In a protocol that fully defined any possible values sent and received, this can sometimes realistically be done, but sometimes not. And if it involves any user provided or external obtained content, it can be impossible to absolutely guarantee.

Sometimes further sanity checks are provided, as discussed in the Sanity Checks section below. But, failing all of that as a guarantee of non-ambiguity, sometimes a system of 'escapement' is used, to force the interpretation of a special value as non-special in a given context. There are various ways of doing this, but it's unlikely to be of value to discuss them since you'd be far better off just avoiding the issue. If you do choose to use a binary protocol, it's almost certain that a header plus payload plus possibly some extra sanity checks will more than suffice to the purpose. We'll examine a sample binary protocol later in this document.

Efficiency

The most efficient protocols are those that include a length indicator, the reason being that those that don't require the receiver to read the bytes one by one until it sees some sort of end marker. The receiver just has read a header's worth of data, verify it using whatever means are provided, then pull out the payload length indicator and issue another single read for that many bytes.

But, on modern hardware, this efficiency argument probably is considered a distant second, or third or fourth, to concerns of simplicity of implementation. Since the number of bytes being transmitted per message (sentence) tends to be smallish, the overhead won't be significant anyway. On a small embedded device perhaps it might be an advantage though if such bulk reads can be processed asynchronously while other housekeeping chores are being done.

Syntax

Once you have decided on the binary vs. textual form of your protocol, and how to properly extract coherent sentences (messages), you then need to deal with the syntax of the message content itself. So we have a block of data, and we've discarded any of the message extraction housekeeping data, and just have the actual message contents, the payload. We need to parse the content of that message payload to see what the other side is trying to say. You could do this many ways, but most protocols end up doing it in the most obvious ways, because they are simple and work well

Here are most of the sentences usually involved:

- **Do This Command.** The controller needs to tell the device to do something, to change the channel, to turn off the light, etc...
- **Give Me Info.** The controller needs to ask the device for something, what's the current channel, what is the off/on state of the light, etc...
- **Here's Some Info.** The device either is responding to a query (previous item above) with the requested info, or it is sending some unrequested notification of something having happened.
- **Configure Yourself.** Some devices provide options in the protocol that affect how the device will respond or act, and therefore the controller needs to send commands to the device to set these options. These are syntactically probably the same as Do This Command, but with just some internal state of the device being the target of the command instead of the physical mechanism being controlled.
- **Acknowledgement.** The device needs to acknowledge that it received a command or request, so it typically sends some kind of Acknowledgement (ACK) that all went fine, or a Negative Acknowledgement (NAK) that it didn't understand what it was sent or couldn't do what it was asked.

This list makes up pretty much all of what transpires between a controller and device. Not a very deep conversation, but an important one. Here are some basic examples of binary and textual syntax. We'll get more deeply into the subject later, but these will serve as a starting point.

Binary Syntax

In a very simple binary protocol, you might have a set of numbers of that represents the commands, e.g. 1 is do a command, 2 is give me info, 3 is here is some info. Another set of numbers represents the things that can be requested or set, e.g. 1 is current channel, 2 is power, 3 is mute, etc...

So the most simple sentence might be the bytes 1,2,0, which might mean, turn off the power because the action is 'do a command', the target is the power state, and the value to set is zero for off. The acknowledgement of setting the state of something might be to just send out the same response as would have occurred if you requested the state, or it might be some simple ACK/NAK specialized message, perhaps with an error response indicator. To request the power, it might be 2,2, which would be a query command and the source being the power. And the response might be 3,2,1, meaning here is some info, the source is the power, and the value is 1, which means On in this case.

Some devices will literally be that simple if they don't have a lot of options. For instance a simple video switcher might have a protocol that simple. Of course there needs to be some more work to be able to correctly recognize the start and end of such messages. But, in terms of the actual message payloads, they can be that simplistic, and of course highly efficient since even with the sentence boundaries overhead you might be only sending four or five bytes.

In those cases where some text needs to be sent in a binary protocol, the protocol will usually either set aside a fixed number of bytes into which the text will be placed, with any unused bytes being set to some known unused text value, like zeros. Or, in some cases it will append the text to the end of the message, where it can be variable length, and will indicate in the fixed binary part of the message the number of variable bytes of text to expect at the end. The latter is often more efficient, but more complex to parse. So, for instance, if the device had a command (value of 4) to display a line of text on its front panel, the command might look like this (assuming our simple text encoding above is used):

1, 4, 3, 4, 1, 4

In this example it is a do a command, and herer we'll create a new 'show text' command with a value of 4, followed by an indicator of how many characters to expect (3), followed by that many characters, in this case the text for DAD. So the first three bytes of the message are the same syntax as above for a 'do a command' message, but in this case the protocol indicates that a show text command will have these trailing text bytes.

- *Of course the commas we are using in these binary examples are for readability purposes only, they wouldn't be part of the actual data, which would just be consecutive binary bytes.*

The ordering of values in a binary syntax generally isn't that important. But, for the sake of those consumers of messages that do not have the ability to arbitrarily examine data within a block of memory, it may be more convenient for them to arrange the data such that the most important stuff comes first, such as the message type. This way, as it moves through the block of memory sequentially and pulls out values, they will see that important stuff first and can often then call other helper methods to parse out the remaining contents based on the type of message.

As discussed below for textual protocols, often a "verb noun value" type of ordering is desirable. The verb is the action to take and generally is the most important, the noun is the target or source of that action and is the next most important, and any modifiers or values are the least important, at least in terms of the natural order of parsing and validation of the contents.

Textual Syntax

A textual syntax might be almost as simple as the above, however whereas a binary protocol might have absolutely no ambiguity as to what the words of the sentences are (in the above example each byte is a word in the sentence, so it's pretty clear), a textual protocol may have to provide some sort of inter-word delimiter so that they can be pulled out of the incoming message successfully. Of course you could have a textual protocol in which each word of the sentence is a fixed length, except perhaps the last one. So you might have some commands like:

```
PWRON
MUTOFF
VOL50
```

In this case, since the values for the action to take are all the same three characters long, there's no ambiguity. You take the first three characters as the command, and the rest of the message is the value, no need to even separate the two, though they often would be for readability by humans. This sort of sentence format is not uncommon in simpler textual protocols. But, if you need to transmit more complex information you may need to deal with more than one variable length word in the sentence. Another common scheme is a comma or space delimited syntax, such as:

```
SET VOLUME -20
GET MUTE
```

Here the syntax is just two or three space separated words. This works well for the most part, however you do have to be concerned if the actual values you send can contain the delimiter character. This is the common problem of 'meta languages' which we will discuss in the next section. In many protocols this is not an issue since all of the data sent and received is defined by the protocol strictly and it can just be arranged that the delimiter will never be used. In some cases only the trailing part can contain arbitrary data so it's not an issue. I.e. you have action and source/target values that are completely defined by the protocol and never have spaces or commas, so they can easily be parsed out by space or comma separation. The value, the trailing part, is then just the remainder of the line, so no parsing is required and it doesn't matter if it contains spaces or commas.

The ordering of the values is completely arbitrary of course, but it's very common in most control protocols to put the action (verb) first, the target or source (noun) second, and any values being sent last. One benefit of it is that the receiver can validate the content of the incoming sentence as he parses through it, because the action to take is the most important aspect of the sentence and it defines what the valid source or target can be. Knowing the source or target tells the receiver what values are valid for the value if any.

Here again, there is really not obvious winner, and the same pros and cons apply here as in the previous section, with binary tending towards more efficiency and being easier to very strictly define, and textual protocols being somewhat more verbose but easier probably to initially understand and possibly easier for modern, structure-less, languages to deal with.

Flow Control

In any conversation, sometimes people just talk too darn fast. The same can apply to controllable devices or automation systems. So there has to be some way to get the other side to slow down where that's required. By using some very simple rules, this problem almost takes care of itself in most protocols, but there are still some issues to consider.

Low Level Flow Control

In some types of protocols, where one side basically does all of the talking, such as perhaps a sensor that just continuously sends out a stream of temperature values, if the sensor can send the values rapidly, it's possible that the receiver could get overwhelmed. That's not too likely in a modern computer based automation system, but it could possibly happen, and potentially more so in an embedded controller that cannot multi-task and therefore has to serially execute all tasks, including reading incoming data.

For such situations, there are two traditional schemes for getting the sender to shut up for a while. One is hardware based and used on devices that communicate through serial ports. There are hardware lines that can be used if both sides support them, in which the receiver can signal to the sender whether it is ready to accept data or not. This is something that is often handled, at least in modern computer operating systems, by the low level serial device drivers, so the application level program doesn't have to worry too much about it.

The other scheme is to do it as part of the protocol itself, so that there are commands that can be sent to indicate that the other side should stop sending, and when it can start sending again. In this case, being part of the protocol itself, the application doing the communications must deal with the details. Though, in text oriented protocols the XON/XOFF special characters (similar to the STX/ETX special characters discussed above) can be used, and the low level serial port support libraries may handle those on behalf of the application as well.

Call/Response vs. Async Notifications

But, having said all of that, don't do any of that. In a typical device control situation, none of that is generally required or even desired. It can all be avoided by the simple expedient of requiring that all transmitted message be acknowledged, explicitly or implicitly. This is pretty easily done by having a protocol with the following characteristics:

- *All outgoing commands from the automation system to the device (to make something happen, set the volume, turn a light on), are explicitly acknowledged by the device, after it has done what it was asked to do, or a negative acknowledgement if it could not do what was asked.*
- *All requests for information from the automation to the device are implicitly acknowledged by the return of the information requested (or a negative acknowledgement that it doesn't have the information requested.)*

In such a system, since the automation system cannot continue to the next step until the device replies, and such a reply is an implicit indication that the device is now ready for another command or request, a natural flow control is established that will insure that neither side talks the other to death. Such a scheme could be referred to by the term 'call and response', since every exchange begins with the controller making a call out to the device, and the device responding. Computer people might refer to it as 'client/server', in that one party (the controller) is a client actively making requests for services, and another party (the device) is a purely passive supplier of services. So the server in this case is very inattentive and only speaks when spoken to.

Many devices use protocols based on this call and response scheme, because of its natural flow control and simplicity of implementation, on both sides. One side only has to send a message and expect a reply. The other side only has to wait for a request and send a reply back. There is a lot to be said for the simplicity of well defined rolls, and where possible this type of protocol should be used.

However, there are some situations where it is not sufficient. In some cases the device must tell the controller important information without waiting around for the controller to ask for it. So you now have a situation in which the client side must both send messages and wait for a reply but also, while waiting for any reply, to be able to process an unsolicited incoming message. This can significantly increase the complexity of the code required to interface to such devices, but the complexity is sometimes necessary for practical reasons. Such notifications are often referred to as 'asynchronous notifications', because they can happen at any time, whereas in a call/response protocol the traffic in each direction is totally synchronized.

Here are some common concerns that drive the selection of a pure call and response protocol vs. an asynchronous protocol.

- **Volume of Data.** If a device only has, say, four pieces of information that can be set or queried by the controller, then there's not much concern. But, if a device has a hundred or four hundred pieces of data, a pure call and response system can be a problem, because the controller cannot know something has changed until it explicitly asks for the piece of information that changed. If there are hundreds of possible pieces of data that could change, the controller has to continually ask for them all the time, making for a lot of traffic almost all of which is redundant.
- **Efficiency.** Again, if there is a substantial volume of data, most of which is not changing most of the time, it's a huge waste of time for both the controller to continually ask for all this data just to have to check it and see if any has changed, and for the device to continually cough up this information and transmit it. For small embedded devices that overhead might be pretty bad.
- **Latency.** Latency in this case is the lag between when some state of the device changes and when the automation system sees that change and can react to it. If the device has a lot of values, the automation controller probably cannot ask for the whole chunk of values multiple times a second. It will have to round robin through the values, asking for either one at a time or chunks of them at a time. This might result in seconds of latency before the controller sees the change, depending on where it is in that round robin cycle when the change occurs.
- **Device Response Time.** Even if the device doesn't have lots of data values for the controller to read, if the response time is quite low, and if the controller's continually banging on the device (to try to keep the latency down) just makes that situation worse, there's still a good argument for having the device send out notifications of changes. Systems like Z-Wave or UPB are typical here, because of the potentially considerable number of devices in their networks and the quite low data rate.

For all these reasons, some devices really need to send out asynchronous notifications of changes. Note that this still leaves some issues that the controller needs to be concerned with, notably:

- **Startup State Acquisition.** Even if the device sends out notifications of changes, the controller will still likely need to get a full report of the state of the device upon controller startup, so that it can cache that data in its own memory. After that it will keep its cached values up to date based on incoming notifications, and possibly some active call/response polling. So using asynchronous notifications doesn't relieve the device from the requirement of providing an efficient means to proactively get all the state information. Don't make the controller spend two minutes just trying to get into sync with the device through slow and laborious single value queries of hundreds of values, before it can come online.
- **Keep Alive.** If a device uses notifications and nothing is changing that requires reportage, the controller might not actually hear anything from the device for long periods of time. So the controller still must periodically 'ping' the device to make sure it's still alive and kicking. So the device should have some low overhead "are you alive" type of query available for this purpose. Alternatively, the device itself can guarantee that it will send out some type of message at least every X number of seconds, with some sort of empty 'ping' type message being used if it has nothing more useful to say. If the controller hasn't seen a message in X seconds, it assumes the device is disconnected and tries actively trying to reconnect and doing the startup state acquisition steps.
- **Missing Notifications.** It's possible that a device might send a notification that somehow gets missed by the controller. Since the controller depends on these notifications, it might never know that it now is providing information that is out of sync with reality. This is often dealt with by using a 'sequence counter' value that is incremented and put into each outgoing message. The receiver maintains a similar counter and if the incoming message doesn't have the expected sequence, something has gone wrong and the connection can be 'recycled' again in order to get back into sync. Such a scheme isn't needed for call/response type protocols, since a lack of response is proof positive of a failure.

So, the bottom line here is that if your device has a fairly small amount of data, and it can either efficiently return it to the controller on a fairly rapid basis or the type of device means that considerable latency isn't so much of an issue then stick with a simple call and response protocol. It's easy to implement on both sides, and even a very simple controller can probably deal with it.

For example, a thermocouple probably doesn't need to be polled rapidly since the temperature just isn't going to change that rapidly. Every five seconds might just fine, or maybe every ten or fifteen seconds. And lots of other devices fall into the category of having a fairly small amount of data and being able to regurgitate it quite efficiently and rapidly without being overloaded. On the other hand, an A/V Processor, an automation panel product that offers lots of analog and/or digital I/O, a lighting system that supports hundreds of loads, etc... these types of products typically will need to support asynchronous notifications of changes, such as the commonly used Elk and Omni automation panels do.

Note that one possible halfway scheme that some devices might choose to implement is to use a call and response system, but provide a special query not for a specific piece of information, but for any bits that have been changed since the last query. So the device would maintain a queue of changes. When the device requests changes, those queued values are sent back and the queue is flushed, ready to start accumulating more changes. And, as always, this still does not relieve the device of the need to support efficient startup acquisition of the device's full status.

And, now we finally get back to flow control again. If the device sends asynchronous notifications, then that means that it is breaking that strict call and response scheme for natural flow control, and it's possible that the controller could get overloaded. As mentioned previously, this isn't very likely on a modern system, based on fast processors and the ability to multi-task. But it might be problematic for some controllers. You may want to deal with this issue by not sending notifications by default, but allowing the controller to explicitly ask for them to be sent, or by providing a configuration setting that can be set by the user via your own configuration interface or DIP switches and so forth.

You might think that you could just require that the controller acknowledge all notifications, but this would lead to a pretty instant death through what is referred to as a 'deadlock'. As soon as the controller sends a request and then waits for a response, at the same time that the device sends a notification and waits for a response, they are now in a dead embrace and neither will be able to respond to the other.

Another consideration if you send notifications is to make it clear to the controller whether a message is an async notification, or a response part of a call and response request from the controller. This makes it much easier for the controller to correctly sort out incoming messages. Some devices send exactly the same thing for an async notification that they would in response to a query for that value. This is convenient since the same code in the controller can handle both. But it can cause the controller to incorrectly think it has seen a response that it is waiting on, when it hasn't yet. So mark responses and notifications explicitly so that they can be distinguished, though everything else about them should probably be the same for convenience.

Acknowledgements

In a way acknowledgements are part of flow control, as mentioned above. As long as the device acknowledges all commands (explicitly) and all requests (implicitly by returning the requested data), a natural flow control is maintained. However, acknowledgements are also part of error reporting. Typically a protocol will have a special positive (ACK) and negative (NAK) acknowledgement message, the latter being returned if the device cannot carry out a requested command, or provide requested information. In such cases, the negative acknowledgement (NAK) can return an error indicator that the controller can report or log for problem diagnosis.

Some devices use as a positive command ACK the same response that they would send for a query of the thing affected. This is not horrible, but probably not optimal either. One reason for doing it might be to work around the fact that the actual value stored in the device isn't exactly what the controller sent. That's not common, and is highly undesirable, but might happen, e.g. a set point that can only be set to even values, so sending 1 would actually set it to 2. By acknowledging the command to set the value with a value query response, the controller adjusts for this quirk in the device naturally since it will then be informed of the actual value set. Generally though a simple ack is more efficient and works as well. The controller already has the value since it just sent it out. It doesn't need to be told about the value again. A positive acknowledgement lets the controller know it can store the value it just set, because that value was successfully stored in the device as well.

In the case of requesting information, typically a controller must be able to respond to either the actual response that it is waiting for, or a NAK response. So it does add a little extra complexity in that case. In the case of an outgoing command to the device, usually an ACK/NAK is all that is expected anyway. In most modern, exception based, languages the low level message reading method can watch for a NAK and throw an exception containing the error info, which means that usually only the highest level interface code needs to actually deal with NAK responses.

Providing good error codes can very much help with in the field problem diagnosis. Many customers will not be capable of breaking out a port sniffer to watch communications and diagnose problems. The controller must of course as well be diligent about reporting such errors, though in some cases only when asked to provide more verbose reporting.

Don't ever have a situation where a controller sends a query or command, and you don't respond if all goes well but you send a NAK if something fails. The controller cannot know whether it should wait or not. All messages from the controller to the device should be either responded to in some way, or not responded to at all (not recommended.) There can't be anything in between because of the ambiguity it entails.

Sanity Checks

It's not uncommon for protocols to include various sanity checking devices so that the recipient of a message can verify that it has clearly been received without any corruption, and that it hasn't somehow gotten out of sync in its attempts to extra full messages from the incoming stream of data from the other side. For some types of devices, controlling critical processes, this can be a very important concern. Some of these sanity checks can be problematic for simple controllers, since they may require doing mathematical calculations that the controller just simply cannot do. To be friendly, it is typical for the device to send them out even though they may get ignored, and to accept messages either with or without them, and to only check them if received.

Checksums

A pretty common and easy to implement sanity check is the check sum. Some of the bytes of the message, such as just the payload part, are just added up, into an 8 or 16 bit value and that sum is transmitted along with the message (in a part not included in the check sum itself of course.) The receiver does the same for the same bytes of the message as received and compares his result with the one transmitted. If they are agree, that's a pretty good sign that the message is correct. It's not a guarantee since it's not completely unrealistic that two arbitrary messages might generate the same check sum, but it's a pretty good check that is simple to do (so even a simple controller might handle it) and efficient to calculate.

Use of checksums really requires a header+payload sort of format to messages, since the checksum itself cannot be part of the bytes that are included in the checksum, or perhaps they are appended to the message in some way.

CRCs (cyclical redundancy checks)

CRCs are similar to checksums except that they don't use a simple sum. Instead they will use an algorithm typically similar to what in software speak would be called a hash. These types of algorithms feed the bits of the message through what is sort of a 'bit blender' where the new bits interact with the previously consumed bits in such a way that it becomes vastly less likely that any two different sequences of bits could result in the same value.

CRCs are obviously much more powerful in terms of ensuring the quality of the transmitted data. But even fairly complex automation controllers might not be able to generate them without allowing for the writing of custom code to do it, because each device might use a slightly different CRC algorithm. There are some standard ones but they aren't universally used, or supported in a canned way in all controllers. So definitely here you would want to make the CRC optional.

As with checksums above, CRCs really requires a header+payload sort of format to messages, since the checksum itself cannot be part of the bytes that are included in the checksum, or perhaps they are appended to the message in some way.

Encryption

This will be discussed below in the section on miscellaneous concerns, but effectively encryption is going to act sort of like a very large hashing operation, in which even a single bit getting scrambled as the encrypted data is transmitted will probably result in a complete or significant mangling of the decrypted data, making it clear it was corrupted in transmission.

Magic Markers

In many binary protocols, magic values are placed into the data, such that the odds of those values appearing by accident if the communications gets out of sync are very low. This is not a particularly strong scheme, but it's often used in addition to one of the other sanity checks and can often allow a bad message to be rejected very quickly without going through the overhead of calculating a checksum or CRC.

Even when a message length indicator is in the message, an end marker can often be used as an extra sanity check. After the receiver sees the message length, it just blindly reads the rest of the message. If it doesn't see the end marker at the end of the read in data, then the message is bad.

Miscellaneous Concerns

In addition to the above material, which covers very obvious and core issues of protocol design, this section will cover some less common ones that you might want to consider.

Encryption

Some devices will require encryption of all messages, both from the controller to it and vice versa. This may seem like overkill, and probably in most cases it is, but for some core devices that have control over your HVAC system, perhaps your security system, it may be something to consider. Note that this is really mostly only an issue for devices that use Ethernet for communications. For devices that use USB or serial connections, those are basically one to one connections from the controller to the device. So there's not much concern that a third party will intercept that information.

For devices that are using the network, anyone with access to the network could potentially tap into that communications and eavesdrop or even take control over those devices. In some cases even here encryption is overkill since if you have someone breaking into your network you have bigger things to worry about. Deal with that for all the obvious reasons and you've dealt in the process with the potential hijacking of your automation devices. But, even so, there are considerations such as the wannabe hacker friends of your children, or the potential for someone hacking into your wireless network through accidental mis-configuration of the router potentially.

In any event, if the device is controlling systems in the user's home such that they could be very concerned about these types of possibilities, you may consider implementing encryption. However, it would almost certainly need to be optional because many controllers will not be remotely capable of implementing typical encryption schemes like AES, Blowfish, etc... The only device supported by CQC that currently required encryption is the HAI Omni automation panel, and that requires a special driver to deal with the encryption requirements.

Login

Short of encryption, you might choose to implement a login feature, where the controller must provide some login information before the device will correspond with it. This is more practical with mediums that have a distinct concept of connected or disconnected. The reason this is important is that the device must know if the connection is lost and go back into 'waiting for connection' mode which forces a new login to establish a 'session' as it is usually called. As long as that session is active, the controller is trusted as assumed to be the same party that provided the login credentials. Serial ports, for instance, don't really have this concept. The only way to know if someone is on the other side is if they respond in the defined way.

General Rules

This section provides some general rules that controllable devices should follow in order to insure a high quality result. The previous discussion has been more about the actual for and flow of communications between the controller and device. But there are other, higher level concerns that are very important and that are very much part of the 'contract' between the two parties to abide by certain rules of protocol. Any deviation from these rules put extra burdens on the controller that must be understood, and that preferably are avoided altogether.

Support Multiple Users

There is no one to one relationship between automation controller and device controlled in a modern, multi-user automation system, such as our CQC product. There are typically multiple users throughout the home, and the automation system is often configured to do things automatically at various times or in response to changes. So there's not a guy sitting in front of a computer pressing a button to see the latest state of a device. In order to support from two to tens or even a hundred consumers of device information around the network, a modern controller cannot go out and talk to the device every time someone needs to know something about its state.

Instead the controller caches its own copy of those aspects of the device's state that it cares to expose, and hands out that cached information to consumers upon request. The controller must keep his copy of the device's state up to date as much as is practical. This leads to two rules that well behaved devices should obey:

- *The device must provide an efficient means to bulk query the state of the device. When the controller starts up, it has to get its cached view of the device in place quickly. Don't require two minutes or querying hundreds of individual values before the controller can come on line.*
- *The device must provide an efficient means to keep up to date on changes in the device. If the device doesn't have much data, then a simple ongoing poll is sufficient, but the device must be able to handle this at a fairly rapid rate without choking. If it cannot, or if it has a lot of data, it should support asynchronous notifications of changes to the controller.*

These issues are discussed in detail in the Flow Control section above.

Device Response Times

The controller is providing a means to get information from and send commands to the device. It is a fairly universal rule that the controller cannot send another query or command to the device until the device has acknowledged that it has completed the last one. If the device doesn't support asynchronous notifications, then the controller has to actively poll the device state fairly rapidly in order to minimize latency (see latency discussion above.) If the device is slow to response, then outgoing commands are often significantly delayed, in order to allow outstanding polling queries to complete first. This causes the automation system to feel spongy and slow.

So the device must provide rapid responses to the controller if the controller is going to provide a high quality experience to the end user. This can be problematic in some cases since many devices cannot multi-task. So the design of the device must not place the needs of the automation system last. It should be designed such that responses can be rapidly provided, or it must provide asynchronous notifications to avoid the issue.

Dead If Off

There are a lot of devices that are of the sort that CQC calls 'dead if off'. This means that they do allow the controller to power them off and on via the control protocol, but that when in the off state they won't respond to anything but a power on command. This is a very bad design for really obvious reasons. All the controller knows about a device is what it can know through the control protocol. A device that doesn't respond to queries is no different from a device that has been disconnected, taken outside, and run over with the lawn mower. From the controller's point of view it doesn't exist.

Therefore, if it doesn't in fact exist, i.e. it has failed because the communications has been severed through some physical or configuration change, or networking problem, then the controller has absolutely no idea until the user asks power the device on. At that point, after the user has sat down with his Banana Daiquiri and bowl of popcorn to watch a movie, the controller tells him it's not

working. This is something that the controller should have been able to tell him within a minute of its having failed. Of course watching a movie isn't a mission critical activity, at least to some people, but other such scenarios could certainly fall into that category.

So please do not design such a device. At the very least, allow it to respond with some sort of "Sorry, I'm powered off right now" response, so that the controller can be sure it's still present and responding. And provide a guaranteed minimum response time by which you will at least provide some sort of response, so that the controller can know that any failure to respond within that period of time is obviously an indication of trouble.

Minimum Inter-Message Intervals

Some devices, even if they use a strict call and response protocol, will still require that the controller respect some minimum interval between messages it sends to the device, often on the order of 100 to 500 milliseconds, sometimes more. This is really a sub-optimal design and to be avoided if at all possible. The call and response scheme should be sufficient to provide the needed flow control.

However, presumably because some devices are both non-multi-threading and don't even have the ability to buffer up incoming bytes from the control interface (serial port, USB port, whatever) in an interrupt driven way, that means that the device must actively be reading the serial port or incoming bytes will be lost. So, even though they have responded to a previous command or query, they now need time to tend to other housekeeping chores before they can get back to looking for control messages again.

If there is such a limitation, it should be kept to an absolute minimum interval, preferably closer to 100 milliseconds than one second.

Sample Protocols

In this section we will examine some real world protocols and discuss their relative merits and deficiencies. Hopefully this will make some of the more abstract and general discussions above a bit more concrete. We'll try to cover a range, from simple to complex and binary to textual. Obviously we cannot completely dissect the more complex protocols, but we can touch on their basic form and flow and what they get particularly right and particularly wrong.

[to be done]