Gilson Embedded Communication Protocol

Version	Date	Name	Comments
A	9/29/2016	Jerry Koepke/Matt Coakley	Initial revision
B	03/27/2019	Matt Coakley	Update document format
C			
D			
E			

Table of Contents

Overview

This document is a specification for Gilson's Embedded Communications Protocol (GECP). The protocol is targeted for an instrument's internal, board-to-board communication as well as the main control protocol between an instrument and the application software.

Notation Conventions

The keywords "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC2119.

Requirements

General Protocol Requirements

- Shall work over any physical interface (RS-232, CAN, USB, Ethernet, etc.)
- Shall allow for easy debug and monitoring of the physical interface (human readable)
- Shall allow for bidirectional, synchronous and asynchronous message flow between all data paths.
- Shall contain only printable ASCII characters in the range of 0x20 through 0x7E.
    - The only allowable exceptions to this requirement are:
        - For the transfer of binary data which is covered in the Sending Binary Data section.
        - For necessary protocol encapsulation used by USB, Ethernet, CAN, etc.
        - Carriage return (0x0D) and line feed (0x0A) characters.
- Shall contain both a carriage return (0x0D) and a line feed (0x0A) character at the end of the message.

Protocol Definition

Message Format

```
?[Sequence,Source,Destination,Type,Mode,Code(MessageData)]?\r\n
```

Every message starts with **?[** and ends with **]?\r\n**. All parameters are comma delimited. The message name and message parameters are enclosed in parenthesis. All message parameters are required to have some value.

Message Fields:

Sequence

The Sequence ID field is a unique number that is used to track a message from initial generation through completion. This is a 32-bit, unsigned value. The value of zero (0) shall be used for messages which are not associated with commands. Once a message is issued, all subsequent return messages related to the initial message will contain this sequence ID of the initial message. Sequence ID numbers should not be reused until a message's flow is complete (all responses, ACK, etc. have been returned).

Source

A unique number used to identify the sender of the command message. This is a 32-bit, unsigned value.

Destination

A unique number used to identify the recipient of the command message. This is a 32-bit, unsigned value.

Type

Indicates the type of message. This field shall be populated with one of the following:

Type	Description

CMD	Command message
RSP	Response message
ACK	Acknowledged message
NAK	Not Acknowledged message
DBG	Debug message
ERR	Error message
STATUS	Status message
DATA	Sampled Data Response message

Mode

Determines how the message should be processed. Only messages of type CMD shall populate this field. All other message types should populate this field with an ASCII ‘0’ (zero). For Command messages, this field shall be populated with one of the following ASCII strings:

Mode	Description
0	Set to a number for any message type other than command
SYN	Subsequent commands are executed once this command has completed
ASYN	Subsequent commands are executed once this command has started
IMD	Command is executed immediately ignoring any running or waiting commands

Code

This field is used for different purposes depending on the Type field:

Type	Description
CMD	Not used, set to 0
RSP	Return code
ACK	Return code of 2
NAK	Return code
ERR	Return code
FAIL	Return code
WARN	Return code
DBG	Not used, set to 0
STATUS	Not used, set to 0
DATA	Not used, set to 0

Return Codes

Code	Description
0	Reserved
1	Success – not typically used, deprecated

2	ACK only – Acknowledgement of a message successful
3	Command completed successfully
4	Device is busy due to other command execution
5	Intermediate/Periodic data message
6	Error on request/sequence ID
7	Invalid destination/device ID
8	Invalid Command Name
9	Command not allowed in this state
10	Receive Timeout error
11	Invalid command parameter
12	Invalid/Missing message start/end tags
13	Command unexecuted due to error
14	Invalid/Missing command start/end tags
15	General warning
16	Invalid/Missing message parameters
17	Command aborted and flushed from the command queue
18	Warning
19	

MessageData

MessageData formats specific to message types:

Type	Format(s)	Description
CMD **RSP** **STATUS** **DATA**	(Name)	**Name** – Executed command. May contain any whitespace between words. No commas or special characters shall be used as part of the Name.
	(Name,P1,P2,…,Pn)	**P1,P2,…,Pn** – Comma delimited parameters. May contain whitespace if necessary, but there shall not be whitespace before or after the commas. Parameters may or may not exist.
	(Name,P1a|P1b,P2a|P2b,…,Pna|Pnb)	**P1a|P1b,P2a|P2b,…,Pna|Pnb -** In the event a given parameter contains multiple pieces of information such as a timestamp and data or a value and units, the data within a given parameter will be delimited with a | character. There is no limit to the number of

	(Name,[< base64DataString >])	pieces of data which can be contained in a single parameter. **base64DataString** - See *Sending Binary Data* section for details on sending binary data within a command.
ACK	(Name or ACK)	**Name** – Executed command. May contain any whitespace between words. No commas or special characters shall be used as part of the Name.
NAK	(Name or NAK)	**Name** – Executed command. May contain any whitespace between words. No commas or special characters shall be used as part of the Name. If the name could not be parsed from the message being responded to, NAK will be used in place of the name.
ERR	(Name,Error)	**Name** – Executed command. May contain any whitespace between words. No commas or special characters shall be used as part of the Name. If the error was not generated as the result of executing a command, “Error Event” will be used in place of the name. **Error -** Error message.
DBG	(Debug,Data)	**Debug-** The string *Debug* will be used for the first parameter at all times. **Data –** Optional debug data.

Sending binary data

If a block of binary data needs to be transmitted within a Command or Response message, the data shall be encoded in base64 and encapsulated within a base64 start/end tag. This is used to pass to binary code packets to support firmware re-flash.

The base 64 data string shall be in big endian format. The following figures show the conversion method from binary to base64 ASCII and the conversion table.

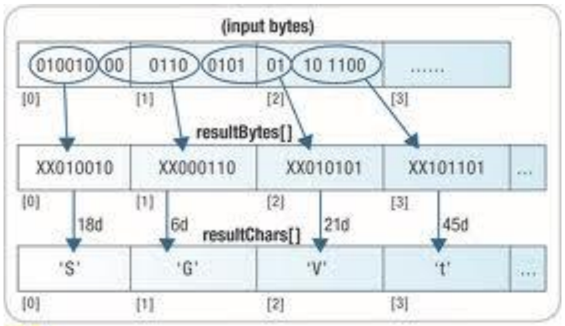
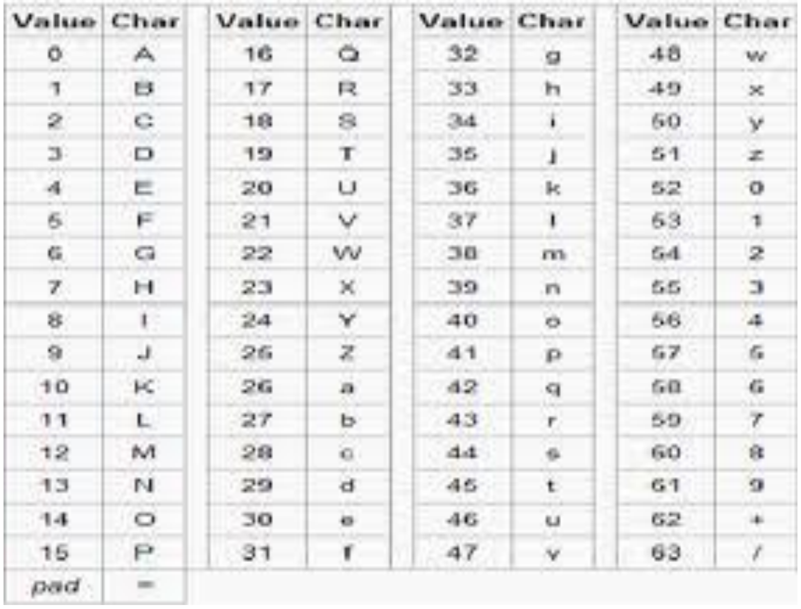


Figure 3: *Base64 encoding mechanism*



Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	W	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/
pad	=						

A base64 encoded data packet shall be encapsulated with a start and end tag to ensure the data is handled correctly. The base64 encoding table will ensure that no tag character sequences will occur in the encoded packet. The encoded packet shall start with a “[<“ and end with “]>”. The size of the encoded packet is limited based on the project needs.

Message Flow

Message acknowledgements

Acknowledgement format:

?[Sequence,Source,Destination,Type,Mode,Code(MessageData)]?\r\n

ACK

All messages, with the exception of ACK and NAK, will be acknowledged by the recipient with an ACK message response. The message data for the ACK will be either the command name associated with the message received, or simply the string ACK. If an ACK or NAK is not received by the message initiator, the original message will be resent by the initiator until an acknowledgement is received or four retries have occurred, whichever occurs first.

ACK example: **?[1000,0,1,ACK,0,2(Get Device ID)]?**

NAK

If data is received which cannot be read for some reason, such as corrupt or incomplete packet, the message recipient will acknowledge the message with a NAK. The sender of the message will resend any message it receives a NAK for. There is no limit to the number of times a message will be resent when a NAK is received. If enough of the message could be read that the associated command name can be parsed, the message data of the NAK response will have the command name in it, otherwise the string NAK will be in the message data portion of the NAK message. If the sequence number can be parsed from the incoming message, that sequence number will be used in the NAK response, otherwise a 0 will be used for the sequence number. The number of

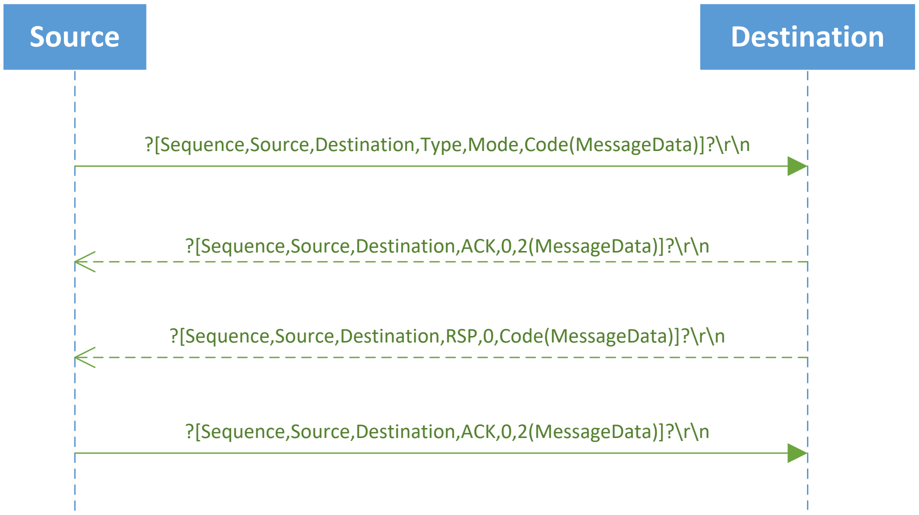
NAK example: **?[0,0,0,NAK,0,14(NAK)]?**

Message Flow

Typical message flow

Under normal circumstances, a message initiator sends a message to a recipient, the recipient acknowledges the message as a valid, performs any actions required, sends a response and the recipient acknowledges the response.

Typical message flow:


Source
Destination
?[Sequence,Source,Destination,Type,Mode,Code(MessageData)]?\r\n
?[Sequence,Source,Destination,ACK,0,2(MessageData)]?\r\n
?[Sequence,Source,Destination,RSP,0,Code(MessageData)]?\r\n
?[Sequence,Source,Destination,ACK,0,2(MessageData)]?\r\n

Immediate Message

Messages sent with mode set to IMD are processed immediately regardless of any other messaged waiting to be processed. The ability to process a specific message immediately will be implementation specific and may not be supported for certain message payloads (i.e. specific commands on an instrument may not support immediate execution). Immediate messages follow the typical message flow.

Synchronous Message

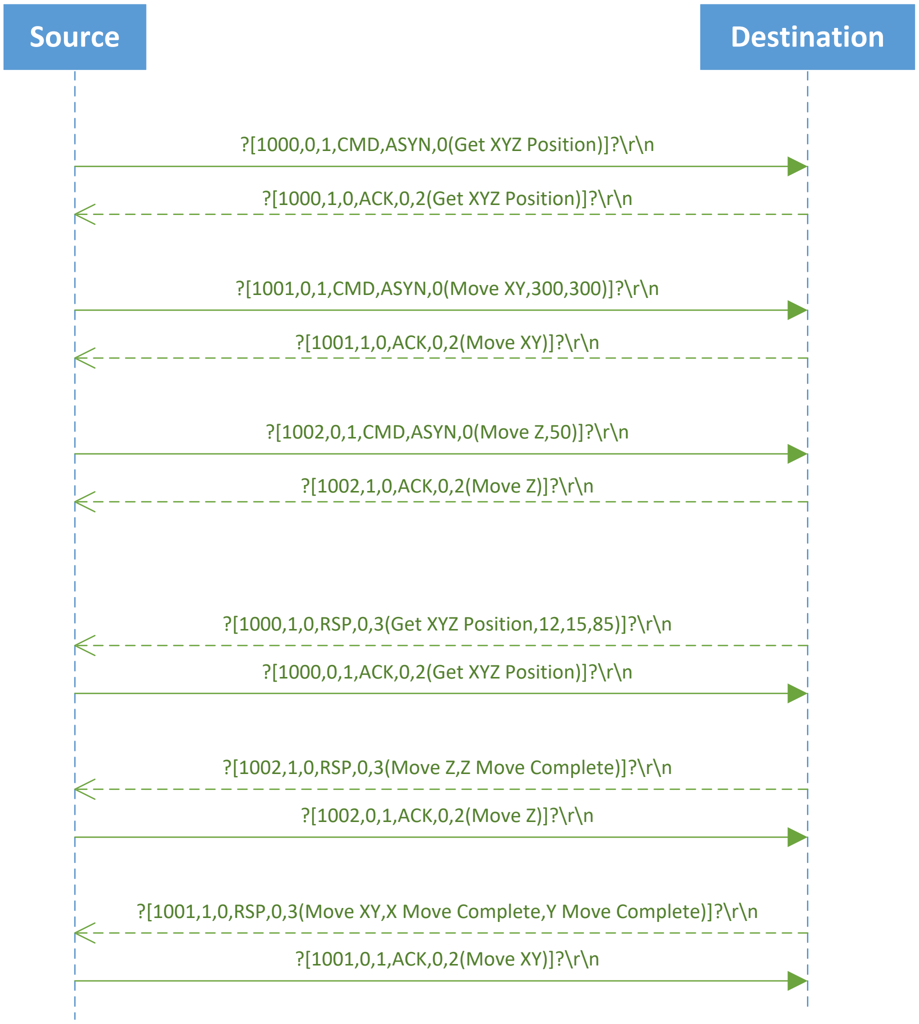
Synchronous messages are processed in the order in which they are received and will not be processed until all previous messages are complete. Responses to synchronous messages are returned in the order they were received. Synchronous messages follow the typical message flow.

Asynchronous Message

Asynchronous messages are processed in the order in which they are received, but are processed in parallel, allowing message processing to start on multiple messages without waiting for completion of the previous message. Because the messages are processed simultaneously, the order in which they complete is dependent on the operation performed and not the order the messages were received. Responses will be sent when the processing for a message is complete regardless of order.

Asynchronous messages follow the typical message flow, but allow for multiple messages to be processed at the same time.

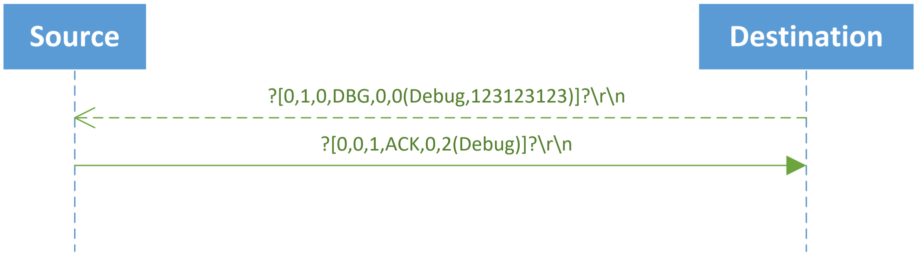
Asynchronous message flow example with responses returned out of order:



```
?[1000,0,1,CMD,ASYN,0(Get XYZ Position)]?\r\n
?[1000,1,0,ACK,0,2(Get XYZ Position)]?\r\n
?[1001,0,1,CMD,ASYN,0(Move XY,300,300)]?\r\n
?[1001,1,0,ACK,0,2(Move XY)]?\r\n
?[1002,0,1,CMD,ASYN,0(Move Z,50)]?\r\n
?[1002,1,0,ACK,0,2(Move Z)]?\r\n
?[1000,1,0,RSP,0,3(Get XYZ Position,12,15,85)]?\r\n
?[1000,0,1,ACK,0,2(Get XYZ Position)]?\r\n
?[1002,1,0,RSP,0,3(Move Z,Z Move Complete)]?\r\n
?[1002,0,1,ACK,0,2(Move Z)]?\r\n
?[1001,1,0,RSP,0,3(Move XY,X Move Complete,Y Move Complete)]?\r\n
?[1001,0,1,ACK,0,2(Move XY)]?\r\n
```

Unsolicited Message

Unsolicited messages may be sent at any time, without a preceding message requesting it. The events which generate unsolicited messages are implementation specific. Unsolicited messages would typically be responses, error, debug, status or data messages.


Source
Destination
?[0,1,0,DBG,0,0(Debug,123123123)]?\r\n
?[0,0,1,ACK,0,2(Debug)]?\r\n

Examples

Note: ← and → denote message direction and are not part of the communication stream

Command

Message exchange for the execution of a typical command:

→ ?[1000,0,1,CMD,0,0(Get Device ID)]?\r\n

← ?[1000,1,0,ACK,0,2(Get Device ID)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

→ ?[1000,0,1,ACK,0,2(Get Device ID)]?\r\n

Data streaming

Message exchange for streaming data:

→ ?[1000,0,1,CMD,0,0(Start Pressure Samples)]?\r\n

← ?[1000,1,0,ACK,0,2(Start Pressure Samples)]?\r\n

← ?[1000,1,0,RSP,0,3(Start Pressure Samples,Success))]?\r\n

→ ?[1000,0,1,ACK,0,2(Start Pressure Samples)]?\r\n

← ?[20,1,0,DATA,0,0(Pressure Sample,12327|22.1,12328|22.0,12329|21.8,12330|21.7))]?\r\n

→ ?[20,0,1,ACK,0,2(Pressure Sample)]?\r\n

← ?[21,1,0,RSP,0,0(Pressure Sample,12331|21.5,12332|21.4,12333|21.3,12334|21.2))]?\r\n

→ ?[21,0,1,ACK,0,2(Pressure Sample)]?\r\n

← ?[22,1,0,RSP,0,0(Pressure Sample,12335|20.9,12336|20.8,12337|20.6,12338|20.4))]?\r\n

→ ?[22,0,1,ACK,0,2(Pressure Sample)]?\r\n

→ ?[1006,0,1,CMD,0,0(Stop Pressure Samples)]?\r\n

← ?[1006,1,0,ACK,0,2(Stop Pressure Samples)]?\r\n

← ?[1006,1,0,RSP,0,3(Stop Pressure Samples,Success))]?\r\n

→ ?[1006,0,1,ACK,0,2(Stop Pressure Samples)]?\r\n

Message NAK

Message exchange for invalid message:

→ ?[1000,0,1,CMD,0,)]?\r\n

← ?[1000,1,0,NAK,0,2(NAK)]?\r\n

→ ?[1000,0,1,CMD,0,0(Get Device ID)]?\r\n

← ?[1000,1,0,ACK,0,2(Get Device ID)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

→ ?[1000,0,1,ACK,0,2(Get Device ID)]?\r\n

Message with no ACK

Message exchange for invalid message:

→ ?[1000,0,1,CMD,0,0(Get Device ID)]?\r\n

← ?[1000,1,0,ACK,0,2(Get Device ID)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

← ?[1000,1,0,RSP,0,3(Get Device ID,VERITY 3011 CONTROLLER,1.0.3.5)]?\r\n

Sending binary data

Message exchange for invalid message:

→ ?[1000,0,1,CMD,0,0(Send Binary Data,[<YWNrIGEgbWVzc2FnZSBmcm9tIGluaXRpYWw>])]?\r\n

← ?[1000,1,0,ACK,0,2(Send Binary Data)]?\r\n

← ?[1000,1,0,RSP,0,3(Send Binary Data,Success)]?\r\n

→ ?[1000,0,1,ACK,0,2(Send Binary Data)]?\r\n