

Common Interface Protocol

Simone Monkton, Development Manager
(403) 212-2192
Simone.Monkton@sage.nelesautomation.com

Murray Peterson, Technical Specialist
(403) 212-2294
Murray.Peterson@sage.nelesautomation.com

Neles Automation, SAGE Systems Division
10333 Southport Road SW
Calgary, Alberta T2W 3X6 CANADA

Abstract

In dealing with the protocols currently in use in our industry, we have all had to deal with the expense, complexity, and risk resulting from some recurring issues. Simple protocols cannot meet industry requirements without non-standard extensions, while the more complex of these protocols are expensive to implement on SCADA host systems, and are difficult (if not impossible) to implement on older and slower RTUs. Also, some modern constraints not anticipated by earlier protocols are: the need for security and encryption on the Internet; provision for the year 2038 rollover; and multiple user access resulting from corporate and utility mergers.

A desirable protocol standard should meet the following minimum requirements:

- Is simple and cost effective to implement*
- Minimizes telecommunication costs*
- Can be retrofitted to existing RTUs*
- Can use pre-existing communication systems without modification*
- Can handle large variations in communication reliability, latency, and bandwidth*
- Is suitable for Front End Processor implementation (e.g. data concentrators)*
- Has intrinsic support for gas, oil, water, and electrical industries*
- Can be easily verified for correctness and completeness*

To meet these requirements, this discussion paper presents a protocol design that includes these features:

- Publish and Subscribe capabilities*
- A minimally layered communications protocol (i.e., the OSI stack is collapsed)*
- Communication is datagram (message) based, not connection (session) based*
- Includes a pass-through facility, which allows implementation of non-standard features without protocol modifications, such as remote configuration and programming of RTUs*
- Data objects are simple and static (custom modifications are discouraged) and capable of supporting multiple industries*
- Includes Host and RTU simulators as part of the specification package to enable complete validation of both host and RTU implementations*

Introduction

Protocols are a major expense in the SCADA industry for a variety of reasons:

Protocol complexity

Larger, more complex protocols require more than 10,000 lines of code to implement. These protocols are obviously expensive to implement, requiring years of programming effort to produce reliable implementations.

Insufficient functionality

Smaller, less feature-laden protocols would seem to be much cheaper to implement, but this is not the case. Since the standard protocol is often incapable of meeting a company's needs, it is soon extended by the RTU vendor or by the company itself. These extensions are outside the protocol definition, so there is an ever-growing list of protocol variations, none of them exactly the same. This means that SCADA vendors constantly must be modifying their protocol implementations to meet the needs of the next variation encountered.

Non-standard features

Most modern RTUs are user programmable, which is both good news and bad to the industry. The programmability allows companies to modify the protocol to meet their needs. However, matching these changes at the SCADA host can be expensive and risky.

Proprietary

There are proprietary protocols that require RTU vendor-supplied libraries or hardware. The libraries never seem to be available for all operating systems and hardware, or they attempt to take control of the telecommunications in addition to the protocol stack. In the worst cases, the SCADA vendor is reduced to reverse engineering the protocol, or setting up a front-end processor just to run the protocol library.

CPU intensive

Replacing a field device is very expensive, so it would be preferable if existing devices could be retrofitted with a newer protocol. However, the larger, more complex protocols require more resources (CPU and RAM) than the older RTUs can provide. Companies wishing to standardize on a single protocol are required to replace all their existing devices, at great expense.

Communications intensive

Some protocols are uneconomical with their usage of communications bandwidth. If a company is being charged for this bandwidth, they will have an excessively high operating cost in addition to the capital cost associated with the protocol implementation.

SCADA upgrades

SCADA systems are frequently modified to improve the product or upgrade technology. Whenever these architectural changes affect the protocol interface, then the protocols must be modified or rewritten to keep pace.

Industry specific

Many protocols are designed to suit the needs of a single industry. This makes it difficult to use in other industries, but in a climate of corporate takeovers and the merging of utilities, this is exactly what the protocol is being forced to do. This generally means further enhancements and modifications by both vendor and customer.

Platform dependent

Some protocols have embedded platform dependencies, either operating system or hardware. These protocols force the purchase of the specific platform, increasing hardware and maintenance costs.

Single point of access

Transportation pipelines deal with custody transfer issues, requiring access to a single RTU by multiple SCADA systems. Protocols that do not support multiple access require each company to purchase, install, and maintain their own RTU at the point of transfer.

Measurement divisions are also inhibited from independent access to the RTU's data, and are required to retrieve it from the host SCADA system.

Business rules

Protocols that embed business rules in their design will continually need modification to match changes to those rules. The more industry specific a protocol becomes, the more likely that it will unknowingly include a few business rules as part of the protocol definition.

Inconsistent time formats

Time zone and daylight savings information may be required at the RTU for a variety of reasons. However, when the protocol implementation is required to handle variations due to varying local time offsets, this just adds more implementation effort without any real benefit.

This paper presents a proposed protocol, the Common Interface Protocol (CIP), which attempts to minimize or eliminate the above problems. To accomplish this, the CIP design includes:

- A minimally layered communications protocol (i.e., the OSI stack is collapsed)
- Datagram (message) based communication as opposed to connection (session) based
- Publish and Subscribe capabilities
- Simple and static data objects which are capable of supporting multiple industries and requirements without changes to the protocol definition
- Binary data objects, which allows implementation of non-standard features, such as remote configuration and programming of RTUs, without protocol modifications
- Time data objects, which enforce consistency and are immune to changes in daylight savings time
- Host and RTU simulators to enable complete verification of both host and RTU implementations

What follows is the rationale behind the design of CIP. The protocol definition itself can be found in Appendix A.

OSI stack

Perhaps the most unusual feature of CIP is its lack of a full 7-layer OSI implementation. SCADA systems deal with communication failures – operators require control over all timing parameters and retries, and they also require notification of all failures and routing changes. The full 7-layer OSI stack is designed to hide these details, which means that protocol implementations must expose the internals of the lower layers to the application layer. This effectively negates one major advantage of using the OSI model, adds extra complexity, and makes plug in replacements for each layer very difficult and/or expensive.

Here is a detailed look at each OSI layer, and how CIP handles (or declines to handle) the requirements of that layer.

Application

The CIP protocol intentionally makes this layer the responsibility of the SCADA host, since this is where business rules and RTU vendor specific requirements are applied. Here are example peer interactions to show the futility of attempting to define this layer within the protocol:

- Gas measurement system downloading gas quality values to a meter run application at the RTU due to a change in a reading from a gas chromatograph being polled by different SCADA host.
- Modification of certain RTU object's permissions due to an operator shift change.
- An external process downloading new RTU configuration information, perhaps even a new version of the operating system.

Presentation

This layer defines the objects that are transferred between the RTU and the host. Data encryption is performed at this layer.

Session

Since the CIP protocol is message based, there is no real session layer in terms of logging on or logging off. A minimal session layer does exist, in that the host sends an initialization message to the RTU, followed by subscription requests, and the RTU publishes the data back to the host.

Transport

The transport layer has been eliminated for reasons of code simplification.

In CIP, message objects are sufficiently small to preclude the necessity of splitting frames into multiple packets. Multiple message objects may be packed into a single message, to allow maximization of bandwidth where the communications channel allows.

CIP does not allow for fragmentation of messages, so message frames and data link packets have a one to one correspondence. Thus, the CIP data link layer is responsible for reliable delivery of all frames/packets.

Network

The network layer has been eliminated due to changing telecommunication infrastructure and business rules.

SCADA systems require that the network layer be handled outside of any protocol, because company business rules apply to network layer transactions. Consider the set-up and use of alternate communication paths to an RTU:

- The initial system uses dial-up connections as an alternate for a failed primary link.
- Several years later, the system is modified to use dial-up connections as the primary path, with a VSAT link for the alternate path.

It is immediately apparent that the protocol cannot, and should not, be responsible for the complex network layer issues involved in a fully redundant SCADA system. If a company requires complex network routing of packets through multiple nodes between the host and the RTU, "off the shelf" solutions are available to cheaply and efficiently perform these functions (e.g. Cisco, Motorola).

Data Link

CIP implements the data link layer in a fairly straightforward manner, using CRCs to guarantee message correctness and a simple retry scheme to guarantee message delivery.

Physical

The physical layer exists, but is not addressed by the CIP protocol. External converters (such as terminal servers) handle the conversion to the desired bit-level protocol.

The CIP protocol is sufficiently configurable to be immune to a wide variation in communications latency, bandwidth and reliability.

Publish Subscribe

The CIP protocol is based on publish and subscribe (pubsub) architecture, using spontaneous report by exception (SRBE). This combination has several advantages:

- Intrinsic support for multiple hosts accessing data from the same RTU.
- The RTU knows exactly what data interests each host. This allows for optimization possibilities at the RTU.
- The communication bandwidth requirements are reduced, since only desired data is sent, and then only when required.
- Communication latency is reduced, since the data is sent to the host immediately by the RTU, without having to wait for a periodic poll by the host.
- Suitable for direct implementation in RTUs, in front end processors (FEPs), and RTU concentrators (i.e. platform independent).

Multi-drop and dial-up systems obviously don't fit into the purest form of PubSub architecture (report by exception, or RBE), so a special *polled* mode was added to the protocol. Thus, CIP supports both host polling and report by exception (but not both at once). The polled mode is designed so that half-duplex and multi-drop connection topologies can be handled.

To enable these modes to initiate properly, the RTU must always start in a quiescent mode, where it is listening for an initialization message from the host (or hosts). The initialization message will indicate the mode of communication to which the RTU must adhere.

In either mode, the basic subscribe and publish methodology is followed. In polling mode, the host still subscribes to a given set of data, but the RTU reports that data to the host only when it is polled. In RBE mode, the RTU is free to send data to the host at any time.

Object model

The primary aim of the CIP protocol object design is simple; allow any data to be retrieved from, or written to the RTU *without changing the protocol code*.

When retrieving data from an RTU, it is convenient to differentiate it in terms of:

- Atomic or composite
- Real-time or historical

All protocols provide mechanisms for retrieving real-time, atomic data without any need for protocol modifications. However, composite objects (such as PIDs, gas meters, tanks) are problematic, since it is common for the elements of composite objects to be modified, added or deleted, according to the company's needs. Any protocol that has been coded to contain "knowledge" of composite data types will need to be modified to reflect changes in these objects. Historical data retrieval is also problematic, since most protocols store this data as composite groupings. Again, any change in this data usually requires protocol modification.

Given the stated requirement of no code changes to the protocol, these are the choices available for handling composite data:

1. Define all possible objects and object variations.
This option is not viable, since any attempt to define all possible variations of a composite object quickly leads to either huge "superset" objects, or an ever increasing list of objects (which require code changes). It is not possible to define all possible objects, since we cannot predict the future.
2. Make the protocol sufficiently configurable to accommodate any possible composite object definition.
In many cases, company business rules apply to composite object definitions, manipulations and interpretations. The cheapest configuration mechanism that can handle all possible business rules is most likely the direct modification of the protocol code (i.e. high risk and very expensive).
3. Eliminate composite object definition from the protocol, and require the SCADA host to map atomic objects into composite objects as desired.
This option moves most of the complexity (such as business rules) to the SCADA host, which is where it belongs. SCADA hosts must deal with this mapping problem already, since some protocols (such as Modbus) do not have composite objects.

CIP uses option 3 above as a guiding principle, and defines as few composite elements as possible, leaving the SCADA host to be responsible for mapping atomic objects into the desired composite data structures. Instead, CIP defines a set of primitive data types, which allow almost any composite to be built at the host level without changing the protocol code. This means that the CIP protocol data objects are simple and static (modifications are not allowed), and it can support any industry specific data object without being modified. Admittedly, the industry support must be implemented at the SCADA host level, but this is arguably where it belongs. If there are multiple hosts accessing data from a single RTU, this also allows each host to implement its own view of the data.

The basic features of the CIP data model are:

- All data objects are named, allowing any desired RTU addressing scheme
- All data objects are time stamped to millisecond resolution
- All data objects have an associated *quality* indicator.

The supported CIP data objects are:

Numeric types

Signed and unsigned versions of long, short, byte, float, double. Used to support discrete and non-discrete field devices.

String type

A length byte followed by data (embedded 0 bytes possible, but discouraged). Used to support events, alarms, and general messages.

Time type

5 bytes indicating seconds since January 1, 1970, Universal Coordinated Time (UTC), followed by 2 bytes indicating milliseconds since the start of the second. Used for all timestamps and RTU time synchronization.

Binary type

Discussed in more detail below. Used for atypical host-RTU interactions.

Historical types

An historical record of any of the previous types (excepting Time). Discussed in more detail below.

History Object

Since we have determined that the CIP protocol will define no composite objects, historical data must, by definition, consist of atomic data types only (float, double, string, etc.). As an example, say we want to store and retrieve hourly history for a meter run, consisting of flow total, maximum flow, minimum flow, and average flow. This data would be stored in the RTU as four history objects, as opposed to a single history object with four elements. The host SCADA system, not the protocol, is responsible for correlating, interpreting and storing these four items.

Binary Type

All RTUs require some mechanism to allow remote configuration and set-up. Rather than attempt to define such a mechanism, the CIP protocol provides a mechanism for external configuration processes to communicate with the RTU by going *through* the protocol, without the protocol needing to know anything about the message contents. CIP defines a Binary data type, which consists of the following elements:

- Routing identifier
An integer value that the SCADA host can use as a key for routing responses to externally generated messages back to the sender.
- Length
An integer giving the length of the binary data
- Binary data
The data itself, with no protocol interpretation

The actual routing mechanism between the protocol and any external process is vendor and operating system specific. A CIP specification defines a standard API (using the C language) for external access to an RTU. This API allows vendors to provide standard RTU configuration programs that will work on any SCADA host using CIP.

Time considerations

A common protocol problem is dealing with time zone and daylight savings time settings at the RTU. The simple solution is to require that all times be in Universal Coordinated Time (UTC). If the RTU requires knowledge of its local time for any reason, this can be accomplished by setting a variable in the RTU to an offset from the standard UTC. However, all time values sent to or from the RTU must be in UTC.

Another issue is the time stamping of historical data periods, such as hourly flow rate. CIP requires that all periodic historical data be stamped at the bottom of the period. E.G. A record which records a value from 12:00:00:001 through to 13:00:00:000 shall be time stamped as 13:00:00:000. Historical records containing instantaneous snapshots (such as time of maximum value within the hour), shall be time stamped at that instant.

For purposes of synchronizing the RTU time with the host time, there shall be a variable of type time in the RTU named "TIME", which may be read by the host as desired. This variable may be read-only in the case of RTUs that set their own time using other means, such as Global Positioning System (GPS).

In order to avoid the year 2038 rollover, the time data type contains five bytes (instead of the usual four) for seconds since January 1, 1970.

Simulators

The CIP specification requires simulators for both the host and the RTU. The host simulator allows RTU vendors to verify the correctness of their implementation, and the RTU simulator allows SCADA vendors to verify their protocol implementation. In a perfect world, the protocol specification document would exactly match the simulator implementation, and in the long run this is expected to be the case. In the short run, any mismatch should be considered as a serious error, and should be resolved with utmost urgency.

A further use of these simulators is to validate the design of the CIP protocol as being correct. It is easy to design a protocol, but much more difficult to guarantee that the protocol design is truly complete. A working simulator assists the designers in identifying flaws in the design.

CIP simulators must be platform independent, and must be freely available to all. To meet these requirements, CIP simulators will be written in the Perl language. Perl is a public domain language, and runs on an extremely wide range of host platforms. The simulators will account for host platforms' byte order, so it can be transported between platforms with no coding changes.

Security considerations

Most existing protocols have few security mechanisms, and even those that do tend to require that the password be sent to the RTU as part of the login message. CIP must have more extensive security provisions for several reasons:

- The CIP protocol is intended to be usable over any communications medium, including public Internet connections. Such public channels provide almost no protection against unauthorized eavesdropping and spoofing.
- Multiple companies may be given access to the same RTU, and it is desirable to limit data on a host by host basis.
- RTU control and configuration must be strictly limited to authorized host systems.

The CIP protocol allows encryption of every message body, using a private key known only to the RTU and the host. To prevent blind message copying, the sequence number is left in the clear, but is used as a "salt" for the encryption. For private lines, where the messages cannot be intercepted by outside parties, the private key may be set to the null string, indicating that encryption is not to be performed on the message body.

At the time of writing this paper, the encryption algorithm has not yet been chosen, but it should meet these minimal requirements:

- "reasonably" fast
- low memory requirements
- capable of implementation on older RTUs
- sufficiently secure to allow internet transactions

- algorithm is not patented or proprietary
- no government imposed export restrictions
- message body need not be padded excessively

A good block cipher candidate appears to be the Blowfish algorithm designed by Bruce Schneier. However, this algorithm requires 32-bit manipulations, which makes implementation on 8-bit processors problematic.

Conclusion

CIP is a proposed protocol standard that has been designed to support multiple industries, and to reduce protocol implementation, maintenance and operating costs. CIP achieves these goals by being simple, flexible, efficient, secure, and verifiable. Simplicity and telecommunications independence are achieved by using a minimally layered, message-based publish and subscribe design. Security is ensured via encryption of the message body. Flexibility is provided through CIP's wide range of atomic object types, including a binary type for atypical host-RTU interactions. Repeatability and verification is assured through fixed object and message definitions as well as by host and RTU protocol simulators.

CIP is currently a preliminary design. Input from interested parties is needed to mature it as a standard. The driving factor behind the design of CIP is the need for a cost effective protocol that can readily support the needs of multiple industries, and is independent of host or RTU modifications.

Disclaimer

The protocol design presented in this paper is presented for discussion purposes only, and does not represent any products under development by Neles Automation or any other company.

References

Mark A. Miller
"Internetworking"
ISBN 1-55851-436-8

Bruce Schneier
"Applied Cryptography"
ISBN 0-471012845-7

Dan Ehrenreich, Motorola
"Operating Benefits Achieved by Use of Advanced Data Communications for Oil & Gas SCADA Systems"
ENTELEC Conference, 1999

Appendix A: Preliminary Protocol Specification

General

- All numeric data are in MSB byte order.
- All items subscribed to “by name”, using the string type.
- All data are time-stamped using time type and include a “quality” descriptor.
- All messages have sequence number (for ACKS and retry schemes)
- RTU can have up to “n” outstanding (unacked) messages to any subscriber (configurable).
- An ACK for sequence number <n> guarantees receipt of all items with a sequence number up to <n> (modulo 65535).

Object definitions

Primitive Object Types

Primitive Name	Object Type Enumeration	Description
Ubyte	0	Unsigned byte (8 bits)
Sbyte	1	Signed byte (8 bits)
Ushort	2	Unsigned short (16 bits)
Sshort	3	Signed short (16 bits)
Ulong	4	Unsigned long (32 bits)
Slong	5	Signed long (32 bits)
Float	6	IEEE 754 float (32 bits)
Double	7	IEEE 754 double (64 bits)
String	8	Length: ubyte Data: <length> bytes of string data
Time	9	5 bytes (seconds since January 1,1970, GMT) 2 bytes (milliseconds since start of second)
Binary	10	Ulong: routing identifier Ushort: Size of binary data (bytes) Data: unspecified except for length An object with contents which are unknown to the protocol driver. The data is preceded by a ulong, which is used by the host as a routing identifier, and a ushort to indicate the data size.

Historical Object Types

These data types are historical records of the previous primitive types (with the exception of Time).

Historical Name	Object Type Enumeration	Description
H_ubyte	100	Historical ubyte record
H_sbyte	101	Historical sbyte record
H_ushort	102	Historical ushort record
H_sshort	103	Historical sshort record
H_ulong	104	Historical ulong record
H_slong	105	Historical slong record
H_float	106	Historical float record
H_double	107	Historical double record
H_string	108	Historical string record
H_binary	110	Historical binary record

Object Format

All objects contained in a message are formatted as follows:

Object Type	Name	Timestamp	Quality	Value
ubyte	e	p	y	as defined by Object Type
	string	7 bytes	ubyte	

Object Type: Object type enumeration as listed in tables for primitive and historical object types.

Name: The name of the object.

Timestamp: Time stamp associated with this object.

Quality: Quality associated with this object.

- 0 = good
- 1 = instrument fail high
- 2 = instrument fail low
- 3 = stale
- 4 = suspect

Note: quality values have not yet been completely defined.

Value: As defined by **Object Type**

Message format

All messages have the following format:

Start Byte	Body Length	Message type	Host Address	RTU Address	Sequence Number	Message Body	CRC	End Byte
'S'	ushort	ubyte	ubyte	ushort	ushort	Body Length bytes of data	CRC32	'E'

Notes:

1. Message body length must be a multiple of 8 bytes for the encryption algorithm to work correctly. Tailing pad bytes must be set to NULL (binary 0).
2. The message body may contain 0 or more data objects.

Message types

These are the allowed message types:

Message Name	Message Type Enumeration	Host-RTU Interaction
Initialize	0	Host → RTU
Subscribe	1	Host → RTU
Change Poll	2	Host → RTU
Integrity Poll	3	Host → RTU
Read History	4	Host → RTU
Write Objects	5	Host → RTU
Read Objects	6	Host → RTU
Publish	7	Host ← RTU
Heartbeat	8	Host ← RTU
ACK	9	Host ↔ RTU
NACK	10	Host ↔ RTU

Initialize

Mode: ubyte

Instructs the RTU what mode of communication will be used (0 = POLLED, 1 = RBE)

Heartbeat frequency: ushort

If the Mode is RBE, then the RTU must send a heartbeat to the host with this minimum frequency (in seconds). The heartbeat is only required if there is no other data to be sent. The host should alarm the remote as being offline when no data or heartbeat message has been received within twice this time period. If the mode is POLLED, then the RTU may use this as a timeout when being polled for data. If the host has not polled for data within twice the heartbeat period, the RTU may discard all change lists and subscription data.

Max message size: ushort

Maximum allowed message body size (in bytes). This allows adjustment for line quality versus bandwidth optimizations.

Max queue size: ushort

Maximum allowable number of outstanding (unacknowledged) messages. Both the host and the RTU must limit themselves to this value. This setting provides flow control and loading control for either the host or the RTU (or both).

Transmit attempts: ushort

Number of transmit attempts to make when sending a publish message to the host (RBE mode only).

Re-transmit delay: ushort

Number of seconds to wait before re-sending an unacknowledged message to the host (RBE mode only).

The RTU must accept this message at any time, regardless of any sequence number mismatches. Upon receipt, the RTU must set both send and receive sequence numbers to 0. The next message sent by the host will have a sequence number of 1, as will the next message sent by the RTU.

The RTU will discard any existing subscription data it has retained for this host, and will discard any outstanding messages to be sent to the host.

RTU responds with an ACK or NACK message.

Subscribe

Name: string

The name of the subscription object.

Type: ubyte

The type of the subscription object. Subscription to a history object does not imply that the RTU is to send all existing history records to the host. Only newly generated history records are to be sent (i.e. records created after this message has been received).

The message body consists of one or more Name/Type pairs, up to the maximum message size as indicated in the Initialize message. A complete subscription list may consist of multiple Subscribe messages. It is not an error to receive duplicate subscription requests.

RTU responds with an ACK or NACK message.

Change Poll

This message has no body.

This message will not be sent to an RTU in RBE mode.

If the RTU is in POLLED mode, then the host must periodically poll for changed data via this request. In POLLED mode, the host will repeatedly poll the RTU with this message until the RTU responds with an empty Publish message or a NACK.

In POLLED mode, the RTU builds a list of Publish messages, and responds with the first message. Subsequent Change Poll messages (with higher sequence numbers) are requests for subsequent messages from the queue. The RTU sends a Publish message with an empty body when there is no more data needing to be sent or a NACK if there are errors in the request. If the RTU receives any message other than a Change Poll during this sequence, it should consider it as a sequence abort, and discard any queued data.

Integrity Poll

This message has no body.

This message requests the RTU to send back the current value of **all** subscribed data items. In POLLED mode, the host will poll the RTU with this message until the RTU responds with an empty Publish message or a NACK.

In POLLED mode, the RTU builds a list of Publish messages, and responds with the first message.

Subsequent Change Poll messages (with higher sequence numbers) are requests for subsequent messages from the queue (and an implied ACK for the current message). The RTU sends a Publish message with an empty body when there is no more data needing to be sent or a NACK if there are errors in the request. If the RTU receives any message other than an Integrity Poll during this sequence, it should consider it as a sequence abort, and discard any queued data.

Read History

Name: string
Name of history object

Type: ubyte
Data type of history object

Start time: time
Time of first element to be retrieved.
If this element does not exist, then the first available one that occurs later in time should be used.

Max number of elements: ubyte
Maximum number of history objects to be retrieved.

The message body consists of one or more of these retrieval requests.

In POLLED mode, the host will poll the RTU with this message until the RTU responds with an empty Publish message or a NACK.

In POLLED mode, the RTU builds a list of Publish messages, and responds with the first message.

Subsequent Read History messages (with higher sequence numbers) are requests for subsequent messages from the queue (and an implied ACK for the current message). The RTU sends a Publish message with an empty body when there is no more data needing to be sent or a NACK if there are errors in the request. If the RTU receives any message other than Read History during this sequence, it should consider it as a sequence abort, and discard any queued data.

Write Objects

Name: string

Type: ubyte

Value: <as defined by **Type** field>

The message body consists of one or more name/type/value triplets.
The RTU responds with an ACK or a NACK.

Read Objects

Name: string

Type: ubyte

The message body consists of one or more name/type doublets.
The RTU responds with one or more Publish messages containing the requested data, or a NACK.

In POLLED mode, the HOST is responsible for limiting the request such that the response message will not exceed the maximum message size.

Publish

This message contains a packed list of objects (as defined under the Object Composition heading). This is the only message used to send data from the RTU to the host.

In POLLED mode, the host implicitly acknowledges receipt of this message by sending a message with a higher sequence number, at which time the RTU may delete the message from its send queue.

In RBE mode, the host will explicitly acknowledge receipt of the message by sending an ACK or NACK to the RTU. Receipt of a NACK indicates that the RTU should delete any subscription data for this host and quit sending any data until an INIT message has been received. The RTU is responsible for retransmission of messages, using the transmit attempts and re-transmit delay values as given in the INIT message. In the case of complete transmission failure, the RTU should delete any subscription data for this host and quit sending any messages until an INIT message has been received.

Heartbeat

Time: time

The time that this message was constructed.

Subscriptions: ubyte

Indicates if the RTU contains any subscription data for the host (0 = no, 1 = yes).

Used in RBE mode only, this message is used to indicate to the host that the RTU is still healthy and that communication channels are still working.

The host will send an ACK or a NACK in response to this message. Receipt of a NACK indicates that the RTU should delete any subscription data for this host and quit sending any messages until an INIT message has been received.

ACK

This message has no body.

An ACK message header will not contain its own sequence number, but will contain the sequence number of the message being acknowledged.

NACK

Reason: string

Indicates refusal to accept a message, and the **Reason** string provides the explanation. Some examples might be:

“RTU is offscan”

“no subscriptions”

“badly formed message”

“crc error”

“Host not authorized”

“Invalid encryption”

“Invalid message size”

“Invalid queue size”

“Item does not exist: <name>”

“Named item has wrong type. Type is: <type>”

The RTU vendor must supply a complete list of NACK strings that the RTU can send back to the host. This is required for internationalization and proper host configuration.

Appendix B: External process API

```
#include <cip_api.h>
```

```
extern int cip_connect (unsigned long routing_id, unsigned short rtu_address, unsigned long connection_id);
```

This must be called before any further operations are allowed.

The `routing_id` is an identifier that uniquely identifies this process. The SCADA host vendor may provide this identifier value.

The `rtu_address` is the address of the RTU you wish to communicate with.

The `connection_id` identifies the communications path to the RTU. The SCADA host vendor will provide this identifier.

The function returns 0 if the call was successful, and a negative value if the call failed.

This negative error code may be translated into a string by the function `cip_get_error ()`.

```
extern const char *cip_get_error (int errorcode);
```

This function returns a pointer to a static string, which is the error message matching the `errorcode` parameter. This value must be one that has been returned from any of the functions in this API.

```
extern int cip_read_configuration (unsigned long routing_id, unsigned short *max_body_size);
```

This function is used to acquire the currently configured value of the maximum allowable message body size.

```
extern int cip_send_binary_object (unsigned long routing_id, cip_binobj *binary_object);
```

This function requests the protocol driver to do the following:

1. Check the provided binary object for correctness.
2. Create a message object containing this binary object.
3. Queue up this message for transmission to the RTU at the earliest opportunity.

The function will return 0 if all three operations were successful, and a negative value if any errors were encountered. The calling application is responsible for ensuring that the amount of data being sent does not exceed `max_message_size`. The RTU is restricted to a single response containing a binary object, which is also limited to the `maximum_message_size`.

```
extern int cip_read_response (unsigned long routing_id, int max_body_size, cip_binobj *binary_object);
```

This function will block until the response message has been received, or the protocol has determined that the response has timed out. If the function returns 0, then the binary object has been copied into the application's binary object variable. A negative return value indicates an error in the function call, or that the RTU has failed to respond.

```
extern void cip_disconnect (ulong routing_id);
```

Informs the protocol driver that the application is disconnecting.

Appendix C: Web Page

Updated versions of the protocol specification and the source code for the simulator can be found at the following URL:

www.sage.nelesautomation.com/public/21conferences/entelec/demo