REACH Protocol Specification

Version 1.4

Link to gDoc source

Table of Contents

Version History

Version	Date	Author	Summary
0	6/14/2023	Wyatt Drake-Buhr	Initial Draft, message pack
1	9/21/2023	Chuck Peplinski	protobufs
1.1	11/17/2023	Chuck Peplinski	Updated device info structure
1.2	11/28/2023	Chuck Peplinski	Described enumerations in parameter info
1.3	12/21/23	Chuck Peplinski	Optional entries in parameter info
1.4	1/4/2024	Chuck Peplinski	Added Time service, updated sizes
2.5	2/8/2024	Chuck Peplinski	Version number to 2.x to reflect common usage. Response replaces reply and result.

Reach version 1.x is generally applied to implementations of Reach that predate this document. This document was versioned 0 for in its first proposal and then incremented to 1.x. As the standard protocol version of Reach is now commonly referred to as “version 2.0”, further versioning of the document will use 2.x.

REACH Access Protocol Requirements

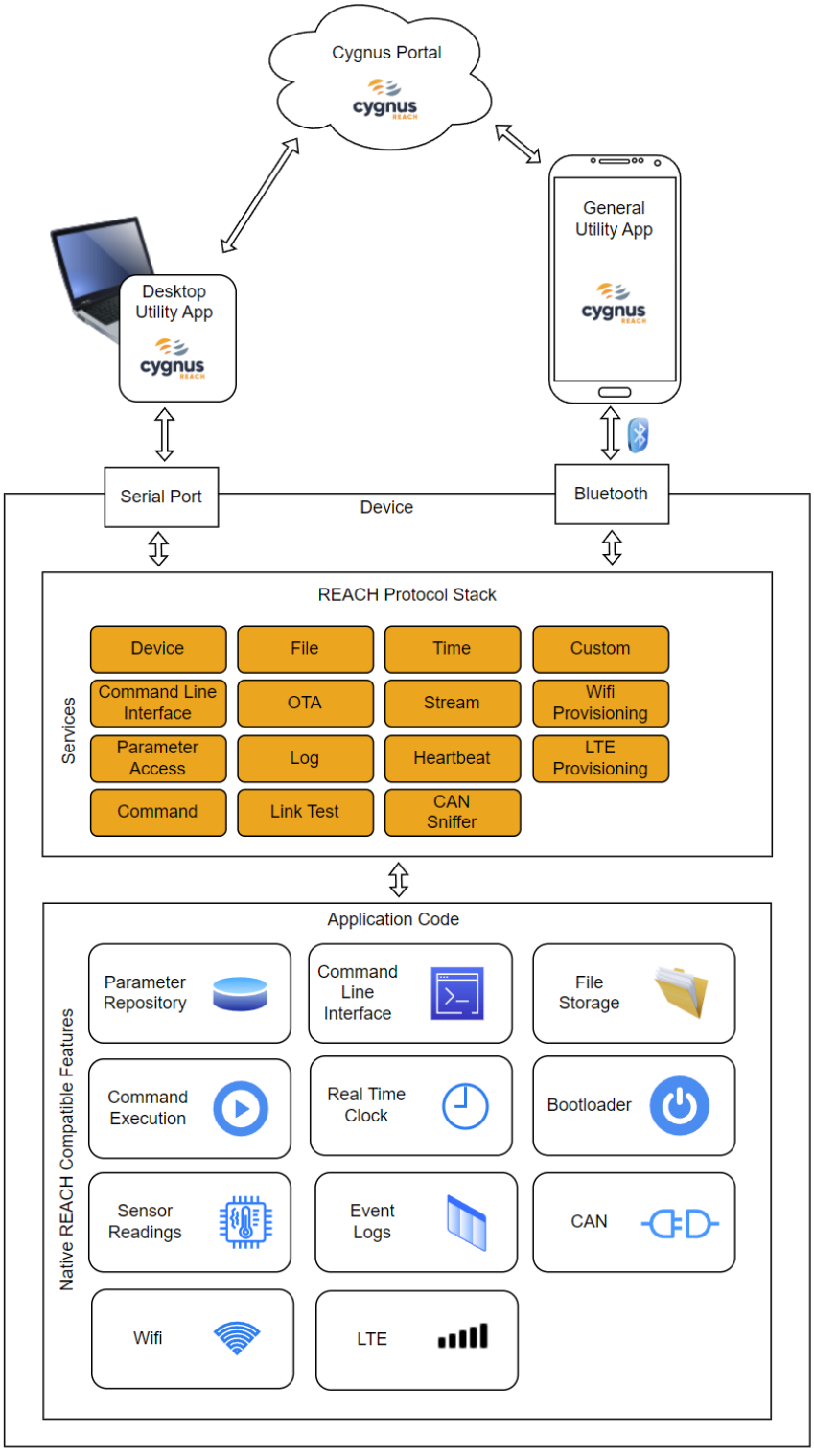
The following are key requirements that guided the direction of this protocol design.

- Seamless integration with Cygnus REACH
- Access device command line remotely
- Read and/or write access to device data
- Support for OTA firmware upgrades
- Efficient bulk data transfer (data log upload)
- Efficient parameter read/write access
- Stream raw sensor data for remote collection
- Execution of Built In Self Test (BIST) on the fly
- Download event logs
- Allow for secure, authorized connections to remote device
- Support embedded devices with multiple “endpoints”. An example might be a Linux machine having an RTOS enabled coprocessor. Each of these might expose a command line and other Cygnus features.
- Allow for complete self-description as well as intended presentation to the Cygnus HMI
- Easily extendable and customizable
- Devolves towards simple - omitting features and definition assumes default modes
- Can be implemented on devices with highly restricted resources (RAM, Flash, MIPS)
- Most common actions are easiest to describe and have the lowest overhead
- Easily ported over many transports - simple Rx/Tx model
- Follows JSON syntax, is human readable

The following existing designs provided inspiration for this protocol:

- Silicon Labs NCP API
- Protobufs
- Nanopb
- Cygnus Device Configuration Model
- Serial Studio
- Cygnus Reach Programmers Introduction

Protocol Overview

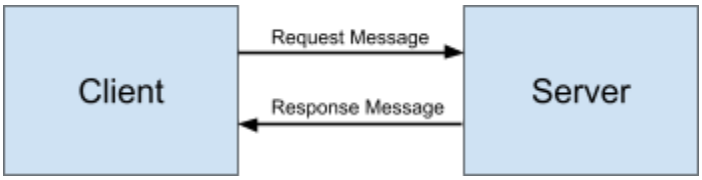


REACH Stack and Device System Diagram

Key attributes and design goals:

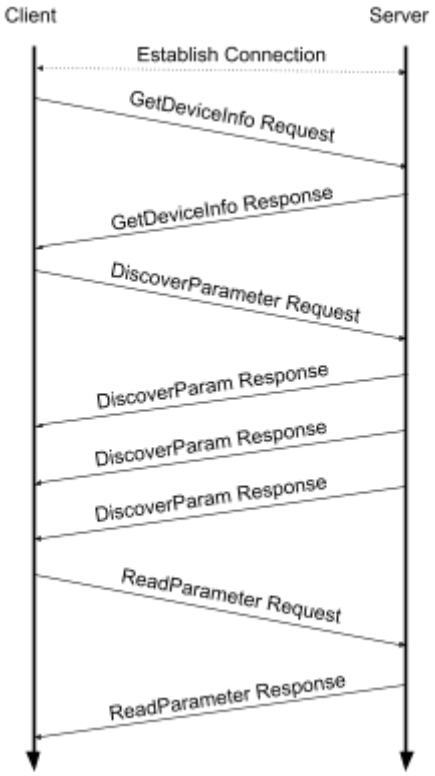
- Server/client architecture (point-to-point)
- Request-response message structure
- Unacknowledged messages allowed (both request and response)
- Organized as collection of Services, which are a collection of Messages, which have one or more supported syntax
- Easily extendable as new functionality is added
- May extend to devices on sub-networks, very simplistic gateway device role is supported
- When presented with a choice, protocol stacks should always default to the simplest format.
- Server side is stateless. The client does not need to maintain or synchronize state data.
- Messages can scale to fit the transport layer payload.

At its core, this protocol assumes a basic client-server architecture where a client issues requests in the form of messages and a server issues corresponding response messages.



In this model, the client is assumed to be the device providing remote support such as a mobile phone and the server is assumed to be the remote IoT device. The message structure is defined in a manner that is transport-agnostic and therefore may be used across any number of networks, but BLE and serial are the primary targets.

A basic message sequence is as follows. It is expected that the client starts every session with a GetDeviceInfo request. This contains basic device information but also informs the client whether its data model needs to be updated.



Example initial message sequence

Bluetooth Low Energy (BLE)

When presented on a BLE interface, the following interface must be implemented. This standard allows for immediate recognition by and compatibility with a Cygnus capable app.

REACH BLE Service:

UUID: edd59269-79b3-4ec2-a6a2-89bfb640f930 (TBD - need Helios / Sun to register for a 16-bit UUID)

Characteristics:

The REACH API characteristic is all that is needed to enable access to the protocol. Client messages sent as individual characteristic writes and server responses are sent asynchronously as notifications.

Field	Description
Name	REACH API
UUID	d42d1039-1d11-4f10-bae6-5f3b44cf6439
Type	String
Length	Variable - up to max BLE APDU
Properties	Write, Read, Notify

Advertisement:
The BLE device must advertise this UUID so that mobile apps can identify it as a Cygnus device.

MQTT

TBD

Protobufs

Because the message size is limited by the BLE frame, we use the .options feature of nanopb to make the size of arrays explicit and to avoid any dynamic allocation.

Continuing Transactions

The Reach “discover” commands are an example of a command that requires more than one message in response. The “discover” prompt and the series of response messages make up a continuing transaction. All of the messages in a continuing transaction share the same transaction_id in the header. The “remaining_objects” member counts down from the initial “number_of_objects” member to zero when the sequence of messages is complete.

Most of the continuing transactions respond with more than one object in a message. The number of objects is determined by the size of the C structures that can be encoded into a 244 byte BLE packet. The server informs the client of the maximum numbers in the device info response.

Message Definitions

Encoding

All messages are encoded using protobuffers.

Conventions

- The message sent from the client is referred to as the “prompt”.
- The message coming back from the server is referred to as a “reply”.
- Messages consist of a “header” and a “payload”.

Schema Example
Communication structures are illustrated here as follows. The code is the ultimate source of truth. This is to illustrate the concept.

```C/C++
// protobuf:
// ERROR_REPORT: Could be a response to a prompt.
message ErrorReport {
  int32  result_value     = 1;    // Error Result
  string result_string    = 2;    // Error String
}

// C Structure
typedef struct _cr_ErrorReport {
    int32_t result_value; /* Error Result */
    char result_string[194]; /* Error String */
} cr_ErrorReport;
```

Data Types

Generally data types are constrained to what is available via protobufs.

Message Header

Reach messages consist of a header and payload. The header can be used for routing, for example, to alternate end points, and for flow control with the remaining objects field.

```C/C++
// protobuf:
message ReachMessageHeader {
  uint32 message_type         = 1;
  uint32 endpoint_id          = 2;
  uint32 number_of_objects    = 3;
  uint32 remaining_objects    = 4;
  uint32 transaction_id       = 5;
}
message ReachMessage {
  ReachMessageHeader header = 1;
  bytes payload = 2;
}

// C Structure
typedef struct _cr_ReachMessageHeader {
    uint32_t message_type;
    uint32_t endpoint_id;
    uint32_t number_of_objects;
    uint32_t remaining_objects;
    uint32_t transaction_id;
} cr_ReachMessageHeader;

typedef PB_BYTES_ARRAY_T(208) cr_ReachMessage_payload_t;
typedef struct _cr_ReachMessage {
    bool has_header;
    cr_ReachMessageHeader header;
    cr_ReachMessage_payload_t payload;
} cr_ReachMessage;
```

The message type is defined in the cr_ReachMessageTypes enum. It specifies the format of the payload. The transaction_id, the number_of_objects, and the remaining_objects members are used when the response to a prompt is too large to fit in one message.

Services

Device Service (required)

This service is required to be implemented by any device that implements the protocol.

GetDeviceInfo

Overview

The device info message provides a starting point for a client to determine the capabilities of the device. This is expected to be among the first messages sent upon initial connection.
Notes:

- The firmware version is an overall version of this endpoint. If a device has more than one firmware load, then the parameter repository is used to share the other versions.
- The parameter_metadata_hash is used to avoid constantly reloading a relatively long list of parameter descriptions. A client is expected to store the hash of the most recent connection to this device and only (re)discover parameters if the hash has changed.
- The application identifier is a 128 bit UUID that can be used to specify a preferred display format on the client. If the device has a custom UI implementation on the remote server or app, then a UUID may be provided in order for the session to automatically associate the custom layout with the device.
- Services are advertised as bits in a bitmask. See the ServiceIds enum.
- Endpoints are advertised as bits in a bitmask. This defaults to zero if there are no more endpoints behind this one. This feature allows a BLE connection to route data to more than one Reach endpoint.
- The sizes_struct in the device info response informs the client of the sizes of the various buffers used by a memory constrained server. These sizes are often set as appropriate for BLE transmission, but these sizes are set by the device and clients should respect them.

Error Handling
Not applicable

Schema Example

```C/C++
// protobuf:
message DeviceInfoResponse {
```

```
  int32 protocol_version          = 1;    // Supported Protocol Version
  string device_name              = 2;    // Name, Typically Model Name
  string manufacturer             = 3;
  string device_description       = 4;    // Description
  string firmware_version         = 6;
  uint32 services                 = 8;    // bit mask
  uint32 parameter_metadata_hash  = 9;
  bytes application_identifier    = 10;
  uint32 endpoints                = 11;   // bit mask
  bytes  sizes_struct             = 20;   // packed

}
```

```
typedef PB_BYTES_ARRAY_T(16)
cr_DeviceInfoResponse_application_identifier_t;
typedef struct _cr_DeviceInfoResponse {
    int32_t protocol_version; /* Supported Protocol Version */
    char device_name[24]; /* Name, Typically Model Name */
    char manufacturer[24];
    char device_description[48]; /* Description */
    char firmware_version[16];
    uint32_t services;
    uint32_t parameter_metadata_hash;
    cr_DeviceInfoResponse_application_identifier_t
application_identifier;
    uint32_t endpoints;
    cr_DeviceInfoResponse_sizes_struct_t sizes_struct;
;
} cr_DeviceInfoResponse;
```

The sizes_struct is a packed C structure as defined in reach_ble_proto_sizes.h. The byte offsets are given in the reach.proto file as SizesOffsets. The buffer_sizes message is provided to help higher level languages unpack the size structure. The size structure informs the client about the limits of the various buffers in the server.

```C/C++
typedef struct
{
    uint16_t  max_message_size;
    uint16_t  big_data_buffer_size;
    uint8_t   parameter_buffer_count;
    uint8_t   num_medium_structs_in_msg;
    uint8_t   device_info_len;
    uint8_t   long_string_len;
    uint8_t   count_param_ids;
    uint8_t   medium_string_len;
    uint8_t   short_string_len;
    uint8_t   param_info_enum_count;
    uint8_t   services_count;
    uint8_t   pi_enum_count;
    uint8_t   num_commands_in_response;
    uint8_t   count_param_desc_in_response;
} reach_sizes_t;
```

In the BLE case these are set to:

```
max_message_size             = 244;  // biggest BLE message
big_data_buffer_size         = 194;  // length string messages
parameter_buffer_count       = 32;   // local parameter store
num_medium_structs_in_msg    = 4;    // parameter read data in one message
device_info_len              = 48;   // device description string
long_string_len              = 32;   // parameter description string
count_param_ids              = 32;   // parameters in one message
medium_string_len            = 24;   // names
short_string_len             = 16;   // units, etc
param_info_enum_count        = 8;    // param ex in one message
services_count               = 8;    //
pi_enum_count                = 8;    // param ex in one message
num_commands_in_response     = 6;    // commands in one message
count_param_desc_in_response = 2;    // parameter info in one message
```

Link Test Service (ping - required)

This service is required to be implemented by any device that implements the protocol.
This service allows testing of the data link.

Ping

Overview
The Ping message allows the client to send an arbitrary data packet to the server and have it echo the contents back. The response includes RSSI information if applicable in order to aid in analysis of wireless connections.

Error Handling
Not applicable.

Schema Example

C/C++

```
// Protobuf
message PingRequest {
  bytes echo_data         = 1;
}
message PingResponse {
  bytes echo_data         = 1;
  int32 signal_strength   = 2;
}

// C Structures
typedef PB_BYTES_ARRAY_T(194) cr_PingRequest_echo_data_t;
typedef struct _cr_PingRequest {
    cr_PingRequest_echo_data_t echo_data;
} cr_PingRequest;

typedef PB_BYTES_ARRAY_T(194) cr_PingResponse_echo_data_t;
typedef struct _cr_PingResponse {
    cr_PingResponse_echo_data_t echo_data;
    int32_t signal_strength;
} cr_PingResponse;
```

Error Reporting

When the Reach stack encounters an error it can post an error message. This consists of an error number and a string. The string can be omitted to save memory. Using this service allows a client to find out about error conditions without monitoring the CLI. Detailed error reporting is encouraged via the CLI service.

Schema Example

```C/C++
// Protobuf
message ErrorReport {
  int32  result_value     = 1;
  string result_string    = 2;
}

// C Structures
typedef struct _cr_ErrorReport {
    int32_t result_value;
    char result_string[194]; /* Error String */
} cr_ErrorReport;
```

Command Line Interface Service (cli)

CLIData

Overview
If the command line (CLI) service is advertised in the device info then the device can act both as a server and a client. As a server it can receive command line prompts from the mobile device. As a client it can emit command line messages at any time. This allows a mobile device to interact with the embedded device at its native command line which often supplies access to a more extensive set of features.

Error Handling
Not applicable

Schema Example

```C/C++
// protobuf
message CLIData {
  string message_data = 1;
  bool is_complete    = 2;
}

// C
typedef struct _cr_CLIData {
    char message_data[194];
    bool is_complete;
} cr_CLIData;
```

Parameter Access Service (pa)

DiscoverParameter

Overview
The DiscoverParameter message provides a means for a client to discover the available parameters and their metadata. This is intended to be called upon the first connection but due to the amount of data possible, it is expected that the client will cache the discovery results and use the hash reported by the GetDeviceInfo message to detect when a new discovery is required.

As with any of the “discover” messages in Reach, the response may be returned as a “continuing transaction”. The ***count_param_desc_in_response*** member of the device info response sizes structure informs the client how many parameter info structures fit into one message.

Enumerations and Bitfields
Parameters can be enums or bit fields. In these cases the parameter description contains a possibly long list of descriptions. An alternate form of the parameter info structure (ParamEx) is used to convey this information.

Challenge Key
The parameter messages include a challenge key. This is intended to provide a level of obscurity when a device is in the field. The default Cygnus Reach app will not be able to provide this key and so a device so protected will provide a limited set of publicly visible parameters. A custom app equipped with the key can go deeper.

Requested List
A parameter info request can specify a set of parameters or by leaving the requested list empty it requests all parameters. Up to 32 parameters (**count_param_ids**) can be requested at once. An empty request will deliver all parameters. When information is requested on multiple parameters, up to two (**count_param_desc_in_response**) can fit in one response message.

Data Types
Supported data types are specified by the cr_ParameterDataType enumeration:

Type	Description
uint32	Unsigned integer
int32	Signed integer

float32	Floating point
uint64	Unsigned integer
int64	Signed integer
float64	Floating point
bool	Boolean
string	ASCII string, null terminated
enum	Enumeration (unsigned integer)
bits	Bitfield
bin	Byte array

Error Handling
If a requested parameter does not exist, an error response will be returned.

Schema Example

```
C/C++
// protobuf
message ParameterInfo {
  uint32 id                          = 1;  // Id
  ParameterDataType data_type        = 2;  // DataType
  uint32 size_in_bytes               = 3;  // Unnecessary?
  string name                        = 4;  // Name
  AccessLevel access                 = 5;  // Access
  optional string description        = 6;  // Description
  string units                       = 7;  // Units
  optional double range_min          = 8;  // Range Max
  optional double range_max          = 9;  // Range Max
  optional double default_value      = 10;
  StorageLocation storage_location   = 11; // RAM or NVM or ?
}

message ParameterInfoRequest {
  uint32 parameter_key          = 1; // Unlock Key
```

```
  repeated uint32 parameter_ids = 2; // ID's to Fetch (Empty to Get All)
}
message ParameterInfoResponse {
  repeated ParameterInfo parameter_infos = 1; // Array of Param Infos

// C
typedef struct _cr_ParameterInfo {
    uint32_t id;
    cr_ParameterDataType data_type;
    uint32_t size_in_bytes; /* Unnecessary? */
    char name[24];
    cr_AccessLevel access; /* Access */
    bool has_description;
    char description[32]; /* Description */
    char units[16];
    bool has_range_min; // range min is optional
    double range_min;
    bool has_range_max; // range max is optional
    double range_max;
    bool has_default_value;  // default value is optional
    double default_value; /* Show instead of value if no value. */
    cr_StorageLocation storage_location; /* RAM or NVM or ? */
} cr_ParameterInfo;

typedef struct _cr_ParameterInfoRequest {
    uint32_t parameter_key; /* Unlock Key */
    pb_size_t parameter_ids_count;
    uint32_t parameter_ids[32]; /* ID's to Fetch (Empty to Get All) */
} cr_ParameterInfoRequest;

typedef struct _cr_ParameterInfoResponse {
    pb_size_t parameter_infos_count;
    cr_ParameterInfo parameter_infos[2]; /* Array of Param Infos */
} cr_ParameterInfoResponse;
```

Schema Example Extended Description of enums and bitfields

```C/C++
// protobuf
// Give names to enums and bitfields
message ParamExKey {
    uint32 id               = 1;  // the valud of the enum
    string name             = 2;  // the name of the enum
}

// also used for bitfields
message ParamExInfoResponse {
    uint32              associated_pid = 1;
    ParameterDataType   data_type      = 2;
    repeated ParamExKey enumerations   = 3;
}
```

The response to the parameter info request can be a mix of ParameterInfoResponse and ParamExInfoResponse. Bitfields and enums will produce the ParamEx data.

ReadParameter

Overview
The ReadParameter message provides a simple way to request the value of one or more parameters. The same caveats about the parameter key and the limit on the number of requested parameters (32, **count_param_ids**) apply to parameter read. When multiple parameters are read a single message can contain up to four (**num_medium_structs_in_msg**) results.

Bitfields and enums are shared in the uint32 member.
Arrays are shared in the string_value member.

Timestamps
The result of a parameter read includes a timestamp. The request to read parameters can be made with “read_after_timestamp”. This would allow the client to request all of the parameters that have not yet been read since this timestamp.

Error Handling
An error report is issued if an error occurs during the read.

Schema Example

```C/C++
// Protobuf
message ParameterRead {
  uint32 parameter_key          = 1;
  repeated uint32 parameter_ids = 2;
  uint32 read_after_timestamp   = 3;
}

message ParameterValue {
  uint32 parameter_id       = 1;
  uint32 timestamp          = 2;
  oneof value
  {
    uint32 uint32_value     = 3;
    sint32 sint32_value     = 4;
    float  float32_value    = 5;
    uint64 uint64_value     = 6;
    sint32 sint64_value     = 7;
    double float64_value    = 8;
    bool   bool_value       = 9;
    string string_value     = 10;
  }
}

message ParameterReadResponse {
  uint32 read_timestamp           = 1;
  repeated ParameterValue values  = 3;
}

// C
typedef struct _cr_ParameterRead {
    uint32_t parameter_key;
    pb_size_t parameter_ids_count;
    uint32_t parameter_ids[32];
    uint32_t read_after_timestamp;
} cr_ParameterRead;
```

```
typedef struct _cr_ParameterValue {
    uint32_t parameter_id;
    uint32_t timestamp;
    pb_size_t which_value;
    union {
        uint32_t uint32_value;
        int32_t sint32_value;
        float float32_value;
        uint64_t uint64_value;
        int32_t sint64_value;
        double float64_value;
        bool bool_value;
        char string_value[32];
    } value;
} cr_ParameterValue;

typedef struct _cr_ParameterReadResponse {
    uint32_t read_timestamp;
    pb_size_t values_count;
    cr_ParameterValue values[4];
} cr_ParameterReadResponse;
```

WriteParameter

Overview

The WriteParameter message provides a simple way to write the value of one or more parameters. The response confirms the completion with an expected result value of zero. Up to four parameters (**num_medium_structs_in_msg**) can be written in one message.

Error Handling

The result value is set to a non-zero value and an error report can be issued if an error occurs during the write.

Schema Example

C/C++

```
// protobuf
message ParameterWrite {
  uint32 parameter_key            = 1;
  repeated ParameterValue values  = 3;      // Array of Write Values
}
message ParameterWriteResponse {
  int32 result                   = 1;      // err~
}

// C
typedef struct _cr_ParameterWrite {
    uint32_t parameter_key;
    pb_size_t values_count;
    cr_ParameterValue values[4]; /* Array of Write Values */
} cr_ParameterWrite;

typedef struct _cr_ParameterWriteResponse {
    int32_t result; /* err~ */
} cr_ParameterWriteResponse;
```

Parameter Notification

Note: Parameter notification is not yet implemented. The timestamp method described under read parameters is an alternative.

Overview

The system can be configured to notify the client when there are changes in parameters.

The ParameterNotifyRegistration message provides a means for a client to register to be notified when a parameter changes or on a regular interval, eliminating the need to poll. Flexibility is provided through several optional parameters:

- Maximum Rate - this can be used to control the rate of updates for frequently changing data
- Minimum Rate - this can be used to ensure a regular update period regardless of value change.
- Minimum Delta - this can be used to regulate updates to value changes of a certain magnitude.

Error Handling

If a requested parameter does not exist, an error response will be returned.

Schema Example

```
C/C++
// Protobuf
message ParameterNotify {
  uint32 parameter_key               = 1;
  bool enabled                       = 2;    // Enabled or Disabled
  repeated uint32 parameter_ids      = 3;
  uint32 minimum_notification_period = 4;
  uint32 maximum_notification_period = 5;
  float  minimum_delta               = 6;
}
message ParameterNotifyResponse {
  int32  result                        = 1; // zero if all OK
  repeated uint32 failed_parameter_ids = 2;
}
message ParameterNotification {
  repeated ParameterValue values     = 2;    // Array of Result Values
}
```

```c
// C
typedef struct _cr_ParameterNotify {
    uint32_t parameter_key;
    bool enabled; /* Enabled or Disabled */
    pb_size_t parameter_ids_count;
    uint32_t parameter_ids[32];
    uint32_t minimum_notification_period;
    uint32_t maximum_notification_period;
    float minimum_delta;
} cr_ParameterNotify;

typedef struct _cr_ParameterNotifyResponse {
    int32_t result; /* zero if all OK */
    pb_size_t failed_parameter_ids_count;
    uint32_t failed_parameter_ids[32];
} cr_ParameterNotifyResponse;

typedef struct _cr_ParameterNotification {
    pb_size_t values_count;
    cr_ParameterValue values[4]; /* Array of Result Values */
} cr_ParameterNotification;
```

Command Service (cmd)

This service exists to allow simple commands to be sent to the device. Examples might include “reboot”, “begin OTA”, or “Reset to factory values”.

Discover Commands

Overview
The DiscoverCommands message returns the definition of the commands supported by the device.

Error Handling
Not applicable.

Schema Example

```C/C++
// Protobuf
message DiscoverCommands {}

message CommandInfo {
  uint32 id     = 1;
  string name   = 2;
}

message DiscoverCommandsResponse {
  repeated CommandInfo available_commands = 1;
}

// C
typedef struct _cr_DiscoverCommands {
    char dummy_field;
} cr_DiscoverCommands;

typedef struct _cr_CommandInfo {
    uint32_t id;
    char name[24]; /* Descriptive name */
} cr_CommandInfo;

typedef struct _cr_DiscoverCommandsResponse {
    pb_size_t available_commands_count;
```

```
    cr_CommandInfo available_commands[6];
} cr_DiscoverCommandsResponse;
```

SendCommand

Overview
The SendCommand message sends a single simple command to the device.

Error Handling
An error is reported if the command value is not supported.

Schema Example

```C/C++
// protobuf
message SendCommand { uint32 command_id = 1; }

message SendCommandResponse {
  int32 result          = 1; // Carries Success / Result
  string result_message = 2;
}

// C
typedef struct _cr_SendCommand {
    uint32_t command_id;
} cr_SendCommand;

typedef struct _cr_SendCommandResponse {
    int32_t result; /* Carries Success / Result */
    char result_message[194];
} cr_SendCommandResponse;
```

File Service

This service exists to allow files to be transferred to and from the device. Files are treated as relatively large blocks of data that must be transferred efficiently (quickly) and accurately. Hence the file service is more complex than most of Reach.

File Access Overview
After files are discovered, the read and write protocols are relatively symmetrical. After an initialization command is issued a series of exchanges follows. The exchange sequence ends with the receiver acknowledging the completion. The number of messages that can be sent without acknowledgement is a parameter of the initialization. With the use of a CRC, this allows for the detection of transmission errors and the resulting retry without waiting for the entire file transfer to complete. Alternatively, BLE transactions are much faster when they are not acknowledged. The protocols are documented in more detail in the Cygnus Reach Programmers Introduction.

GetFileDescription

Overview
The GetFileDescription message returns the definition of the files supported by the device. Up to two files can be described in one message. A continuing transaction is used to describe more files.

Error Handling
Not applicable.

Schema Example

```C/C++
// Protobuf
message DiscoverFiles { }

message DiscoverFilesResponse {
  repeated FileInfo file_infos = 1; // Array of File Infos
}

message FileInfo {
  int32 file_id                     = 1;  // ID
  string file_name                  = 2;  // Name
  AccessLevel access                = 3;  // Access Level (R/W)
  int32 current_size_bytes          = 4;  // size in bytes
```

```
  StorageLocation storage_location    = 5;
}

// C
typedef struct _cr_DiscoverFiles {
    char dummy_field;
} cr_DiscoverFiles;

typedef struct _cr_FileInfo {
    int32_t file_id; /* ID */
    char file_name[24]; /* Name */
    cr_AccessLevel access; /* Access Level (Read / Write) */
    int32_t current_size_bytes; /* size in bytes */
    cr_StorageLocation storage_location;
} cr_FileInfo;

typedef struct _cr_DiscoverFilesResponse {
    pb_size_t file_infos_count;
    cr_FileInfo file_infos[4]; /* Array of File Infos */
} cr_DiscoverFilesResponse;
```

File Transfer Init

Overview

A file transfer is opened using the FileTransferInit message from the client. The continuing transaction can require many messages. max_bytes_per_message is communicated from the server in the GetDeviceInfo exchange. The messages_per_ack member affect the efficiency of the transfer.

Error Handling

Reports an error if a file ID is unknown, permission violation, or invalid operation (access request past end of file).

Schema Example

C/C++

```
// Protobuf
```

```
message FileTransferInit {
  uint32 file_id                = 1;    // File ID
  uint32 read_write             = 2;    // 0 for read, 1 for write.
  uint32 request_offset         = 3;    // where to access in the file
  uint32 transfer_length        = 4;    // bytes to read or write
  uint32 transfer_id            = 5;    // In case of multiple transfers
  uint32 max_bytes_per_message  = 6;    // From GDI
  uint32 messages_per_ack       = 7;    // num messages before ACK.
  uint32 timeout_in_ms          = 8;    // ms before abandonment
}
message FileTransferInitResponse {
  int32 result                  = 1;    // 0 if OK
  string error_message          = 2;
}

// C
typedef struct _cr_FileTransferInit {
    uint32_t file_id;
    uint32_t read_write;
    uint32_t request_offset;
    uint32_t transfer_length;
    uint32_t transfer_id;
    uint32_t max_bytes_per_message;
    uint32_t messages_per_ack;
    uint32_t timeout_in_ms;
} cr_FileTransferInit;

typedef struct _cr_FileTransferInitResponse {
    int32_t result;
    char error_message[100];
} cr_FileTransferInitResponse;
```

File Transfer Data

Overview
The WriteFile message allows a server to write part of a file. The offset is needed on the first message to set the internal file index, but may be omitted on subsequent calls.

Error Handling

Both the TransferData message and its notification contains a result which can be non-zero if an error has occurred. Message exchanges can also time out.
The correct response to a non-zero result is to terminate the transfer session so that the initiator can possibly recover and restart.

Schema Example

C/C++

```
// Protobuf
message FileTransferData {
  int32 result                  = 1;    // non-zero for error
  uint32 transfer_id            = 2;    // Transfer ID
  uint32 message_number         = 3;    // counts up
  bytes message_data            = 4;    // Data
  int32 crc32                   = 5;
}

message FileTransferDataNotification {
  int32  result                 = 1;    // err~
  string error_message          = 2;
  bool   is_complete            = 3;
}

// C
typedef PB_BYTES_ARRAY_T(194) cr_FileTransferData_message_data_t;
/* Bi-Directional Message */
typedef struct _cr_FileTransferData {
    int32_t result; /* non-zero for error */
    uint32_t transfer_id; /* Transfer ID */
    uint32_t message_number; /* counts up */
    cr_FileTransferData_message_data_t message_data; /* Data */
    int32_t crc32; /* Optional crc for integrity checking */
} cr_FileTransferData;

typedef struct _cr_FileTransferDataNotification {
    int32_t result; /* err~ */
    char error_message[100];
```

```
    bool is_complete;
} cr_FileTransferDataNotification;
```

EraseFile

Overview
The EraseFile message allows a server to erase the contents of a file. Not yet supported.

Error Handling
Reports an error if a file ID is unknown, permission violation, or invalid operation.

Schema Example

```C/C++
// Protobuf
message FileEraseRequest {
    uint32 file_id                = 1;    // File ID
}
message FileEraseResponse {
    uint32 file_id                = 1;    // File ID
    int32  result                 = 2;    // err~
    string error_message          = 3;
}

// C
typedef struct _cr_FileEraseRequest {
    uint32_t file_id; /* File ID */
} cr_FileEraseRequest;

typedef struct _cr_FileEraseResponse {
    uint32_t file_id; /* File ID */
    int32_t result; /* err~ */
    char error_message[194];
} cr_FileEraseResponse;
```

Time Service

The time service provides an ability to get and set the time known by an embedded device. This service allows access to the device RTC. The time service carries time as a signed 64 bit number of “UTC” seconds in the Linux epoch (since 1970) as well as a timezone offset. Experience shows that this factoring of the time allows for a globally unique time in the UTC field along with an offset that allows the display to indicate an accurate local time.

GetTime

Overview
The GetTime message allows the client to read the current time on the device.

Error Handling
The result will be zero if there is no error.

JSON Schema Example

```C/C++
// Protobuf
message TimeGetRequest {}

message TimeGetResponse {
  int32 result              = 1; // Carries Success / Result
  string result_message     = 2;
  int64  seconds_utc        = 3; // Linux Epoch
  optional int32  timezone  = 4; // Seconds offset
}
// C
typedef struct _cr_TimeGetRequest {
    char dummy_field;
} cr_TimeGetRequest;

typedef struct _cr_TimeGetResponse {
    int32_t result;         // Carries Success / Result
    char result_message[194];
    int64_t seconds_utc;    // Linux Epoch
    bool has_timezone;
    int32_t timezone;       // Seconds offset
} cr_TimeGetReponse;
```

SetTime

Overview
The SetTime message allows the client to set the current time on the device.

Error Handling
The result will be zero if there is no error.

JSON Schema Example

```C/C++
// Protobuf
message TimeSetRequest {
  int64  seconds_utc        = 1; // Linux Epoch
  optional int32  timezone  = 2; // Seconds offset
}

message TimeSetResponse {
  int32 result          = 1; // Carries Success / Result
  string result_message = 2;
}

// C
typedef struct _cr_TimeSetRequest {
    int64_t seconds_utc;    // Linux Epoch
    bool has_timezone;
    int32_t timezone;       // Seconds offset
} cr_TimeSetRequest;

typedef struct _cr_TimeSetResponse {
    int32_t result; /* Carries Success / Result */
    char result_message[194];
} cr_TimeSetResponse;
```

Not Yet Implemented

Version 0 of the Reach protocol specified several more services that are not yet undertaken. The plans for these are briefly noted in the following sections.

Stream Service

The stream service is intended to support freeform data flowing from one device to another. This would be in the form of ongoing notifications. This remains a planned feature. It is not expected that the device maintains any copy of this data. The transport is not error checked. It is effectively a means to notify the other end with changing data.

Stream Data Formats

A stream is delivered as raw bytes. It is up to the application to decide on the nature of the bytes. The generic Cygnus Reach application can record a stream to a file.

Schema Example

```C/C++
// Protobuf
message DiscoverStreams {}

message DiscoverStreamsResponse {
  repeated StreamInfo streams = 1;
}

message StreamInfo {
  int32 stream_id             = 1;
  AccessLevel access          = 2;    // Access Level for Stream, R/W.
  string name                 = 3;    // Name of Stream
  int32 max_bytes_per_message = 4;    // Max Size of Messages.
}

message StreamOpen {
  int32 stream_id             = 1;    // Stream ID
  AccessLevel access          = 2;    // Read or Write.
}

message StreamOpenResponse {
    int32 stream_id           = 1;    // Stream ID
```

```
    int32 result              = 2;    // Carries Success / Result
}

message StreamClose {
    int32 stream_id           = 1;    // Stream ID
}
message StreamData {
  int32 stream_id             = 1;    // Stream ID
  uint32 roll_count           = 2;    // Message Number (Roll Count)
  bytes message_data          = 3;    // Data
  int32 crc32                 = 4;    // Optional for integrity checking
}
```

OTA

The OTA service is conceptualized to support “over the air” firmware updates. All IoT devices should support a convenient firmware update. Many of the SoC’s used for IoT systems support their own optimized update solution. When this is the case the product specific mobile app (and web app) should support such a solution.
There are also cases in which the SoC standard solution is not enough for the product. This is particularly true when there are multiple CPU’s in the system with one communication link. The authors of Reach have a solution for this case.

WiFi Provisioning

There remains a case to provide a common way to configure the WiFi in an embedded device. This service allows a device to search for and join a Wifi network, assumed to be a separate link from this interface.

GetWifiStatus

Overview
The GetWifiStatus message allows the client to read the current state of the Wifi link.

Error Handling
Not applicable.

JSON Schema Example

```python
# Client Request to Wifi status
"gwifi": null

# Server Response
"gwifi~": {
  "state": "Connected",
  "ssid": "i3 Guest",
  "security": "WPA2"
  "rssi": -30
}
```

DisconnectWifi

Overview
The DisconnectWifi message commands the client to disconnect from the current AP. This has no effect if currently disconnected.

Error Handling
Not applicable.

JSON Schema Example

```python
# Client Request disconnect from a wifi AP
"dwifi": null

# Server Response
"gwifi~": null
```

ScanWifiAP

Overview
The ScanWifiAP message commands the client to scan for available access points within range to which it can connect.

Error Handling
Returns an error if currently connected to an access point and thus unable to scan.

JSON Schema Example

```python
# Client Request to scan for wifi Access Points
"swifi": null

# Server Response
"swifi~": [
  {
    "ssid": "i3 Guest",              # SSID
    "security": "WPA2",              # Security mode
    "rssi": -20,                     # Receive Signal Strength
    "channel": 1,                    # Wifi Channel
    "mac": "ca:d7:19:d8:a6:44"       # MAC address of the AP
  },
  {
    "ssid": "i3",
    "security": "WPA2",
    "rssi": -40,
    "channel": 5,
    "mac": "ca:d7:19:d8:a6:44"
  },
  {
    null                              # Signifies the end of the list.
  }
]
```

ConnectWifiAP

Overview

The ConnectWifiAP message commands the client to connect to a specific access point.

Error Handling

Dependent on the connection operation.

JSON Schema Example

```python
# Client Request to connect to a specific wifi Access Points
"cwifi":   {
  "ssid": "i3 Guest",              # SSID
  "password": "Parmenter",         # AP Password
  "mac": "ca:d7:19:d8:a6:44"       # (O) MAC address of the AP
```

```
}

# Server Response
"cwifi~": null
```

Others

The Reach system can expand to support other services. Proposed examples include “LTE provisioning”, “Heartbeat”, “CAN” and “Custom” services. There is no immediate plan to build these into Reach.