Cygnus Reach Programmers Introduction

I3 Product Design
Version 0.91
Date December 29, 2023

Executive Summary

Cygnus Reach is a system to provide customer support to users of embedded IoT (Internet of Things) devices. This support is also very useful to developers of such systems.

In case you aren't familiar with Cygnus Reach, here are two links with some overview.

- https://cygnustechnology.com/
- https://cygnustechnology.com/see-reach-in-action/

This package demonstrates the code in the device. It's a small part of the Cygnus ecosystem. This document describes the design of the Cygnus Reach IoT device platform as implemented on a common Silicon Labs SoC. A companion "Getting Started" document provides more introduction.

Audience

This document is addressed to the embedded developer who is considering using the Cygnus Reach system in a product.

Document Purpose

This document is provided to introduce programmers to the Reach stack as implemented in the Thunderboard demonstration project.
The Cygnus Reach concept has some history at i3 where it has been deployed in a series of products. This version is effectively a rewrite from the ground up. It is designed to be deployed more widely with less customization required.

The authors assume that the audience has some familiarity with the Cygnus Reach concept. Cygnus Reach includes features such as video and screen sharing that are quite independent from the embedded device. This document is about the embedded device portion of the system.

Product Vision

The Reach stack enables the embedded system to be in contact with a mobile device over BLE for support and for development.

- Cygnus Reach is about supporting embedded devices. The Cygnus Reach vision is to be a framework that is easily deployed into a range of products. This requires several supporting components that are part of a larger ecosystem.
- Customer support technicians are one user of Cygnus Reach. But the product vision is for the Cygnus Reach device info interface to be of critical utility to the embedded

developer. Hence Cygnus Reach is not bolted on to an existing product but rather built in from the ground up.
- The embedded developer must be able to expose the capabilities and debugging features of a device without creating a special web page. The vision is to enable the embedded developer to construct the device info view without involving app or web developers. This recommends a generic “display” client and an efficient data description mechanism.

Overview

A typical Cygnus Reach system has at least two components:
- An embedded device acting as a “server” for its “device info” page.
- A web page or an app with a rich user interface acting as a “client” to display the device info page.

A mobile device may act as a “relay” to connect the embedded device to a web client. The mobile device could also be the client on its own.
Bluetooth Low Energy (BLE) is most commonly used to communicate with the supported device, but other interfaces are possible if they are more appropriate.

The embedded server device and the display client must share a common language. This is the Reach Protocol, described in a separate document. The interface is defined using “protobufs” to leverage the rich feature set supplied in multiple languages.

Reference implementations of embedded systems are available. The Thunderboard demo runs on Silicon Labs very small BLE demo board. The OpenPV reference version runs on an Enovation display. Both talk to Android and iOS mobile devices.

The key features of the implementation include:
- Organization into pages that are convenient for users.
- A standard way to identify the device and its firmware version(s).
- A “parameter repository” to access control and status variables using a key-value pair model.
- Access to a command line interface
- A “file access” mechanism that supports high rate transfers of larger blocks of data.
- A simple means to issue “commands” to the device.

The two demo systems illustrate Reach on systems of different complexity. The Enovation display runs Linux and is an example of Reach in a sophisticated system. The Thunderboard demo shows Reach with no OS as appropriate for tiny embedded systems. Reach can certainly be deployed in everything in between.

The three system approach to support (device, phone, web page) is a key concept. This acknowledges the three users in the system. The product user has access to an embedded device as well as a mobile phone. The support engineer uses a web browser so as to display

data on a larger screen. The embedded system designer is the third user, as the embedded system builder must include the support required.

A proper Cygnus Reach system design includes much input from customer support engineers. The Cygnus Reach system exists to facilitate field support. Hence the information presented to the support engineers must be what they need. With new products the embedded system designer often has no customer support engineers available to provide design guidance. Hence Cygnus Reach systems are often deployed in two phases. The first phase exposes the things that the embedded system designer finds necessary. The second phase tunes this to emphasize the things customer support needs to see first.

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:

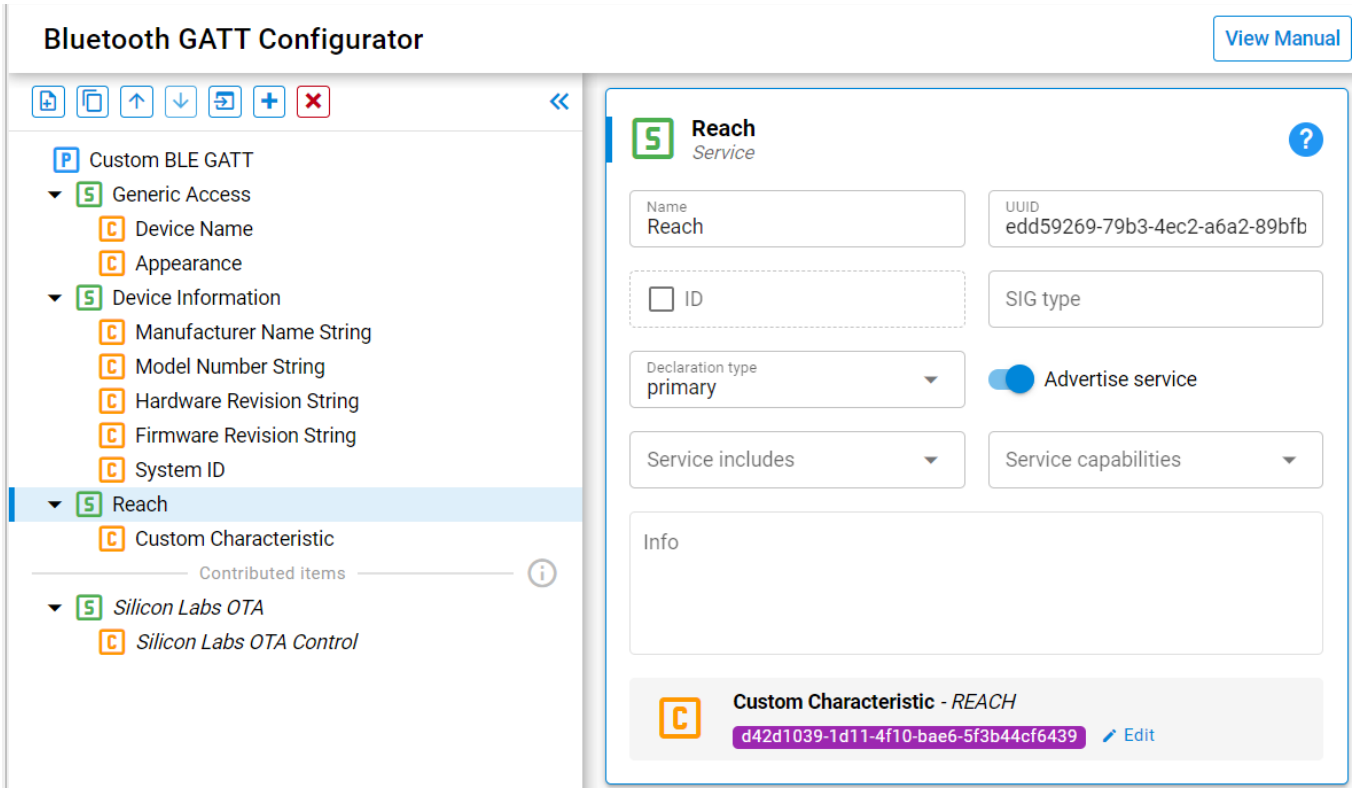
UUID: edd59269-79b3-4ec2-a6a2-89bfb640f930

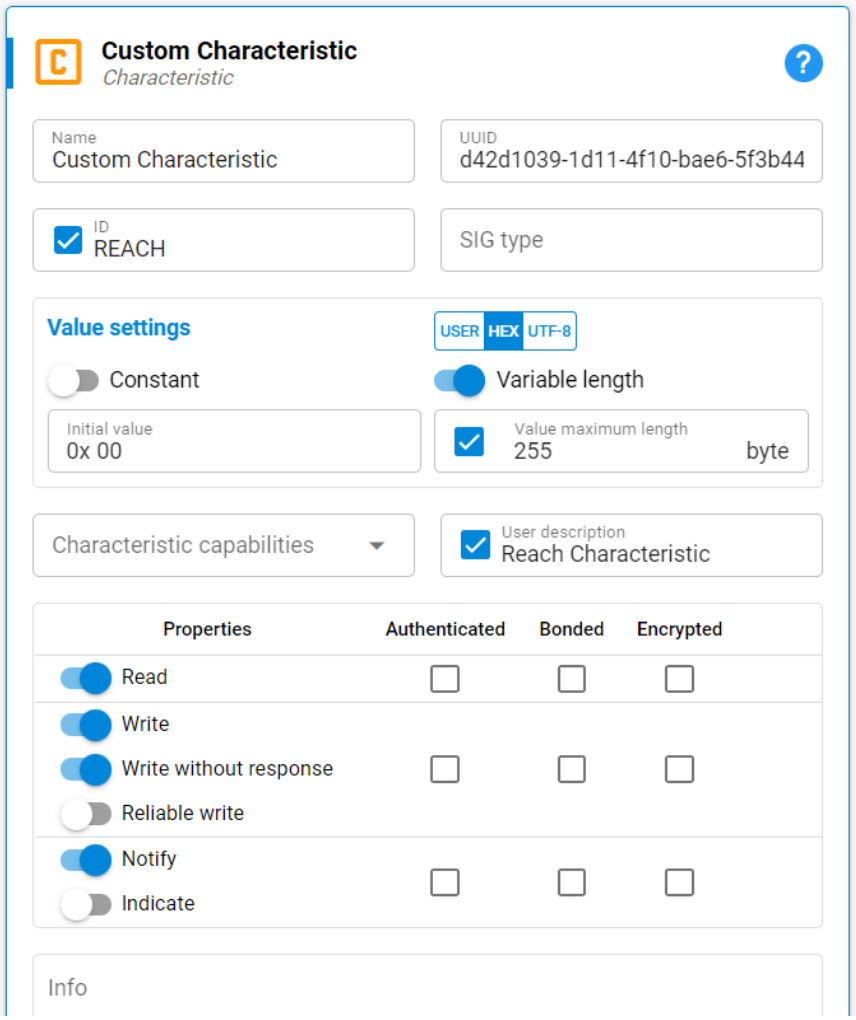
Characteristics:

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. Note that the “Generic Access” service must also advertise to support long device names.


Bluetooth GATT Configurator
View Manual
Custom BLE GATT
Generic Access
Device Name
Appearance
Device Information
Manufacturer Name String
Model Number String
Hardware Revision String
Firmware Revision String
System ID
Reach
Custom Characteristic
Contributed items
Silicon Labs OTA
Silicon Labs OTA Control
Reach
Service
Name
Reach
UUID
edd59269-79b3-4ec2-a6a2-89bfb
ID
SIG type
Declaration type
primary
Advertise service
Service includes
Service capabilities
Info
Custom Characteristic - REACH
d42d1039-1d11-4f10-bae6-5f3b44cf6439
Edit


Custom Characteristic
Characteristic
Name
Custom Characteristic
UUID
d42d1039-1d11-4f10-bae6-5f3b44
ID
REACH
SIG type
Value settings
USER
HEX
UTF-8
Constant
Variable length
Initial value
0x 00
Value maximum length
255
byte
Characteristic capabilities
User description
Reach Characteristic
Properties
Authenticated
Bonded
Encrypted
Read
Write
Write without response
Reliable write
Notify
Indicate
Info

System Structure

The demo has two parts, namely a “server” written in C and a “client” written in Kotlin or Swift or typescript. The demo server runs on a Silicon Labs (SiLabs) Thunderboard. It advertises itself as a Reach device on BLE. Android and iOS mobile apps are available as the demo client. These are available in the corresponding play/app store. Cygnus also supports a web client.

The SiLabs demo is available on the Cygnus Technology github site (https://github.com/cygnus-technology)

The code can be seen as three parts. The “app” is specific to the product, which is the demonstration here. A protobuf module defines the Reach protocolo and is shared with other systems. The C language “Reach stack” (reach-c-stack) is reusable code that implements the Reach protocol with an appropriate interface to the app. The entire set is built to port easily to other systems, whether in the SiLabs ecosystem or other BLE centric systems like those from Nordic. The C-stack includes two files that target the SiLabs BLE interface. These use a small and well defined interface to the rest of the stack.

How to build and run the demo is described elsewhere in the “Getting Started” document.

Porting the Demo

We encourage you to run the demo as is on the Thunderboard as it gives you a concrete reference. This section outlines the process of porting the code to another system. Here I assume it is a C project.
The demo uses no RTOS. Everything goes through an event loop which is part of the SiLabs BLE architecture. Reach could easily be broken off into a separate RTOS task, but this is not demonstrated.
The recommendation in general is to bring up your application so that it behaves just like the Reach demo, and then go on to customize it for your own usage. The rest of this document attempts to give some background to better understand the application.
The demo application can be configured to print out the “wire” traffic in the form of the bytes sent over BLE. This can be helpful when porting.

Begin by getting your BLE to work. Get your system to advertise the Reach characteristic.
Next drop in the proto and c-stack parts of the project. These should compile.
In the apps directory, the files that are prefixed with “reach” may need to be changed for your system. The other files there can be thought of as a dummy database which you can use for testing.

Directory Structure

The “reach-tboard” project uses this top level directory structure.

Reach-c-stack

The directory “reach-c-stack” is also designed to be used unchanged. The “silabs” directory in there supports working on Silicon Labs systems, but this is easily replaced if not appropriate. The sources of all of this are provided. Subject to all the usual caveats (don’t break things!) you can change these to debug and optimize. It’s the protocol on the wire that must be matched.

- It provides the portable core of reach. A user should not have to change this. It could be provided as a library but we choose to provide it in source form.
- Includes the “weak” functions that the app must implement.
- Includes an IoT-Core directory with support functions for logging which rely on printf().
- Includes a “lib” directory containing open source cJSON files.
- Includes a “silabs” directory containing two files that target the stack at the SiLabs BLE interface.
- Maintained as a git submodule to be reused by other projects.

reach_proto

The directory “reach_proto” is the protobuf source. Along with the .proto file there is a script that creates a .options file that matches the device code for BLE. You can use the prebuilt C code for protobufs if you don’t want to bother with this script. Nothing else in this directory should need to change.

- Includes a “proto” directory which includes the .proto source files.
- Includes ansic/built directory with the generated .pb.c/.pb.h files. These are provided so that users do not have to regenerate the files. The procedure to regenerate the files is found in the readme.md file.
- Maintained as a git submodule to be reused by other projects.

App

The directory “app” defines the specific behavior of the demo. You can reuse some of this, such as the “t1_params.c” file to simplify the bringup. But any real project will likely change these files. Functions that override the default “weak” crcb functions should be defined in this directory.

Application Structure

The Reach embedded system is written in C specifically to remain attractive to very small and simple embedded systems. A number of design decisions follow from this.

1. The core Reach stack, which is provided by i3, should be reusable without changes on multiple platforms.
2. The core Reach stack relies on a set of callback functions that must be implemented by the target application. We refer to these as “the weak functions” because they are implemented using the gcc “weak” feature. This allows us to avoid the function pointers which add complexity to debugging. These functions are prefixed with “crcb_”
3. Since C has no formal namespace support the reach stack is implemented in one file so as to keep private variables static. This hides these static “member variables” from global access.
4. Malloc is not used. Large data structures are not instantiated on the stack. The buffers that are used to code and move data are statically allocated.
5. The .options features of nanopb are used to statically define the memory used for protobuf handling.
6. The .options file is generated using an H file. This allows an application to change the buffer sizes. The sizes are tuned for BLE.

Event Loop Structure

From the point of view of an application designer, the reach code is all executed in a single “cr_process()” function which must be called with some regularity. The process function will typically exchange one message with the client over BLE. The process function is called with a tick count, which would typically be milliseconds since system start so that the reach stack can include support for timed notifications. But the details of how this is implemented in the customers system are very flexible. A system with an RTOS might run Reach in a task. A system without an RTOS might call the process function in its loop. The reach stack calls back into the application using the “weak” functions. The embedded system must override these weak callback functions to provide the reach stack with what it needs.

Feeding Data

Data can be pushed into the stack or pulled from the surrounding application. The BLE code pushed coded data into the stack by calling cr_store_coded_prompt(). The socket based test harness overrides the implementation of crcb_get_coded_prompt() to accept data from a socket. The weak default implementation of crcb_get_coded_prompt() assumes that cr_store_coded_prompt() has been called.

A similar dichotomy exists with the coded reply. The socket based test app calls crcb_send_coded_response() which is overridden to send to a socket.

Client - Server Architecture

Reach generally states that the embedded device is a server and the phone is a client. This implies that the server only responds to requests from the client. The sort of asynchronous notifications required to support a remote command line go against this flow. This has not been a problem in BLE systems, but when we test using a socket link we find that we must create a second socket pair oriented in the opposite direction.

An easy way to think about this is to consider the device as a server for most of the Reach protocol. Notifications are an exception because there the device acts more like a client. While the bulk of the Reach protocol is initiated by the mobile device as a client, notifications are initiated by the embedded device. An easy way to handle this is to create an independent communication path for the notifications generated by the device.

Examples of “notifications” that come to the mobile device without prompt are:

- Error reports
- Remote command line
- Parameter notifications

On Logging

Printf style logging and a command line interface are fundamental to embedded development. The reach system relies on the logging support features that are documented here. The reach system provides a rich set of support for these features. In addition to the serial port traditionally used, reach supports a command line interface over the BLE connection. Other solutions could be substituted if appropriate.

Requirements

The authors worked off of this set of requirements:

1. Users must be able to enable and disable logging on a per-function basis. This must be configurable via the command line at run time. Log calls are provided with a “mask” to serve this purpose.
2. “Levels” separate from masks are unnecessary. The typical hierarchy of error, warn, debug, info, etc. is too limiting. Masks take this place and masks must be per function.
3. A “\n” line ending is provided by default, but it must be possible to print into the buffer “bare” without this termination. LOG_MASK_BARE allows for this.
4. Log entries classed as “error”, “warning” and “always” cannot be disabled by the mask. Errors are printed red. Warings are yellow. Always is white.
5. Function and line are not included by default. Users add them where appropriate.
6. Reach defines a set of masks that can be used to debug its internal functions. Other mask bits are reserved to user applications.
7. It must be possible to turn off all logging at build time to minimize size.
8. It must be possible to access the CLI remotely.

Implementation

All of this is supplied by i3_log.h/.c. As provided, iI3_log.relies on printf. This can be ported as necessary. More information on the remote CLI is provided in a subsequent section.

The “lm” command (for log mask) enables the user to see more or less logging. Commanding “lm” alone shows the settings. The log masks are defined in i3_log.h:

```
//  The lowest nibble is reserved to system things.
#define LOG_MASK_ALWAYS     0x01    // Cannot be suppressed
#define LOG_MASK_ERROR      0x02    // Prints red, cannot be suppressed
#define LOG_MASK_WARN       0x04    // Prints yellow, cannot be suppressed
#define LOG_MASK_BARE       0x08    // trailing \n is omitted
#define LOG_MASK_REMOTE     0x10    // Set this to indicate that a message
should be shared remotely.

// These used by Reach features.  Enable them to debug and understand.
#define LOG_MASK_WEAK       0x20    // print in weak functions
#define LOG_MASK_WIRE       0x40    // show what is on the wire
#define LOG_MASK_REACH      0x80    // show reach protocol exchanges
#define LOG_MASK_PARAMS     0x100
#define LOG_MASK_FILES      0x200
```

Using Protobufs

We use nanopb to convert the .proto file into C structures. Following nanopb guidelines, we use a .options file to avoid the use of malloc(). All arrays are converted to fixed sizes which are set in the .options file. The .options file is generated using a python script, reach_proto\proto\preprocess_options.py. This reads in reach-c-stack/reach_ble_proto_sizes.h and a prototype of the options file and outputs an options file that reflects the sizes set by the device. The sizes here are optimized for efficient BLE transfer. A system that does not use BLE could adjust these sizes. The UI applications are designed to respect these size constraints that are advertised in the device info structure.

Protobuf Version Handling

The device information structure includes a protocol_version member. The device populates this with cr_ReachProtoVersion_CURRENT_VERSION which comes from the proto file. The mobile or web app functioning as the reach client compares this value to the value that it found in the .proto file. They should be the same. The protobuf concept allows quite a bit of flexibility in the way of adding things to the interface. Consider two use cases:

1) A product in the field might implement new features in a firmware update. The system should be aware of this from the firmware version in the device info structure. But a system designer might decide to change something significant in the protobuf file. This version can be used to choose two code branches to support the two versions.
2) While this version should not change in release, an application in development might see different versions of the proto file. It is expected behavior that the client will compare the protocol version it reads from its local .proto file with the protocol version received from the server. Warn the user if these two do not match!

On Memory Allocation

The thunderboard reach system does not rely on malloc. All memory allocation is static. The protobuf system uses dynamic memory allocation in languages like Java that have garbage collection. Using dynamic allocation in C puts the responsibility on the caller to free the memory and the authors wish to avoid this.

Memory Usage

The Reach thunderboard application version 3.1.9 was analyzed for memory usage by sorting the output of the “nm” tool. These numbers are approximate. The code was compiled with -Os for size. These values are intended to give an order of magnitude overview of memory usage. Consider in particular the memory requirements that scale with the number of parameters.

- The Reach demo code with protobufs occupies about 17k bytes in flash
    - 4500 bytes for the primary Reach library

    - 2330 bytes for the callbacks.
    - 5190 bytes for the protobuf library code
    - 1988 bytes for the protobuf initialization data
    - 1364 bytes for the Reach interface to the SiLabs BLE stack.
    - 348 bytes for the i3 log functions
    - 1920 bytes for the CLI functions
    - When “NO_LOGGING” is not defined about 14k are added for strings.
- Further flash describes parameters, commands and files:
    - 180 bytes per parameter
    - 28 bytes per command
    - 40 bytes per file

- Reach data requires about 2k in RAM, mainly for communication buffers.
- Parameter values occupy 56 bytes of RAM each.

Communication Buffer Structure

Reach conceptually uses six buffers to exchange prompts and replies. Each buffer is nominally 244 bytes, matching the BLE buffer size. The third buffer points to the first buffer as it can easily be reused. A further reduction of one buffer could be accomplished by encoding the payload directly into the outer message structure, but this makes the code rather confusing so this optimization is not shared.

1) sCr_encoded_message_buffer
2) sCr_uncoded_message_structure
3) sCr_decoded_prompt_buffer (shared with encoded message buffer)
4) sCr_uncoded_response_buffer
5) sCr_encoded_payload_buffer*
6) sCr_encoded_response_buffer*

Received encoded prompt message to be decoded.
sCr_encoded_message_buffer

Decoded message
cr_ReachMessage sCr_uncoded_message_structure
Contains the encoded prompt payload to be decoded

Decoded prompt to be processed.
sCr_decoded_prompt_buffer
Reuses the sCr_encoded_message_buffer.

Raw payload data to be encoded
sCr_uncoded_response_buffer

Payload encoded.

sCr_encoded_payload_buffer

Payload could be encoded directly into sCr_uncoded_message_structure saving a buffer at the cost of added complexity.

Raw message to be encoded

Encoded payload currently copied into **sCr_uncoded_message_structure**

Encoded message to be transmitted

sCr_encoded_response_buffer

A separate pair of “ping pong” buffers are used to encode notifications.

Threading concerns

All traffic flows through the cr_process() function. It is called from one thread. Hence the Reach system is built to be single threaded and no mutex protection is necessary. The remote CLI support adds some complexity in that it can produce data at any time. This is why it uses separate communication buffers.

Services

Reach devices advertise that they support a set of “services” such as “parameters”, “files” and “commands”. Reach devices can implement as many or as few services as are appropriate.

Files, Parameters, and Commands

Parameters are supported using a simple key-value pair structure. The decision to avoid dynamic memory allocation causes the parameter storage structure to have a fixed and limited size. The limit is set to 32 bytes for a string. Data that is too large for this is easily handled using the “file” construct. Reach provides no formal file system. Files can instead be thought of as a key-value pair in which the value size is not limited. While the parameter structure is optimized to fetch several parameters together, the file structure is optimized to transfer a larger block of data as quickly as possible. “Files” can be mapped to a file system if that is appropriate for the application. They can also simply be larger blocks of data maintained by the app. Commands provide a simple means to remotely trigger a function with fixed parameters. Commands could always be implemented in other ways. An example is the command to enable the remote CLI. This can also be engaged using the command line, but providing a command makes it easier to get started with these features.

Multi-Message (Continuing) Transactions

The response to DISCOVER_PARAMETERS, and in fact to any “discover” command could extend over multiple “messages”. To define terms:

- A *transaction* is a series of messages.
- A *message* has a *header* and a *payload*.
- The *prompt* is a received payload.
- The *response* is a generated payload.
- When transferring a file there is a further entity known as the “*transfer*”. The file (read) is a *transaction*. It can be made up of a series of *transfers*, each terminated by an acknowledgement.

The response to any “discover” message may require several messages to complete the transaction. For this purpose the cr_process() function must call a handle_continued_transactions() function before looking for a new prompt. The reach system must keep track of any continuing transactions.

Each discover handler first checks whether the request is null. The request will be valid if this is a new request and it will be null if called for a repeating request.
The system stores a “continued_message_type” as well as at “num_remaining_objects”. The handle_continued_transactions() function first looks at the continued_message_type. It will be invalid when there is no continuing transaction. If there is such a transaction pending the continued_message_type tells us what it is. The appropriate handler function produces a payload which is encoded and we exit the cr_process() function with the encoded message.

The various messages in a multi-message transaction are tagged with the same transaction_id in their header. This is helpful when checking for timeouts in the communication. Each transaction has a defined timeout and its completion is easily determined.

File transfers are made in a series of transactions. A transfer_id is included in the payload to tie this series of transactions together.

Read File Sequence

Reading a file exercises a sequence of transactions. The sequence is tracked by a state structure in the stack. The application must simply provide a function to read the data.

(Phone sends) Transfer Init

- File ID, read/write, size, offset,
- Messages per ACK, Transfer_id

(device responds) Transfer Init Reply (error if not allowed)

Repeat:

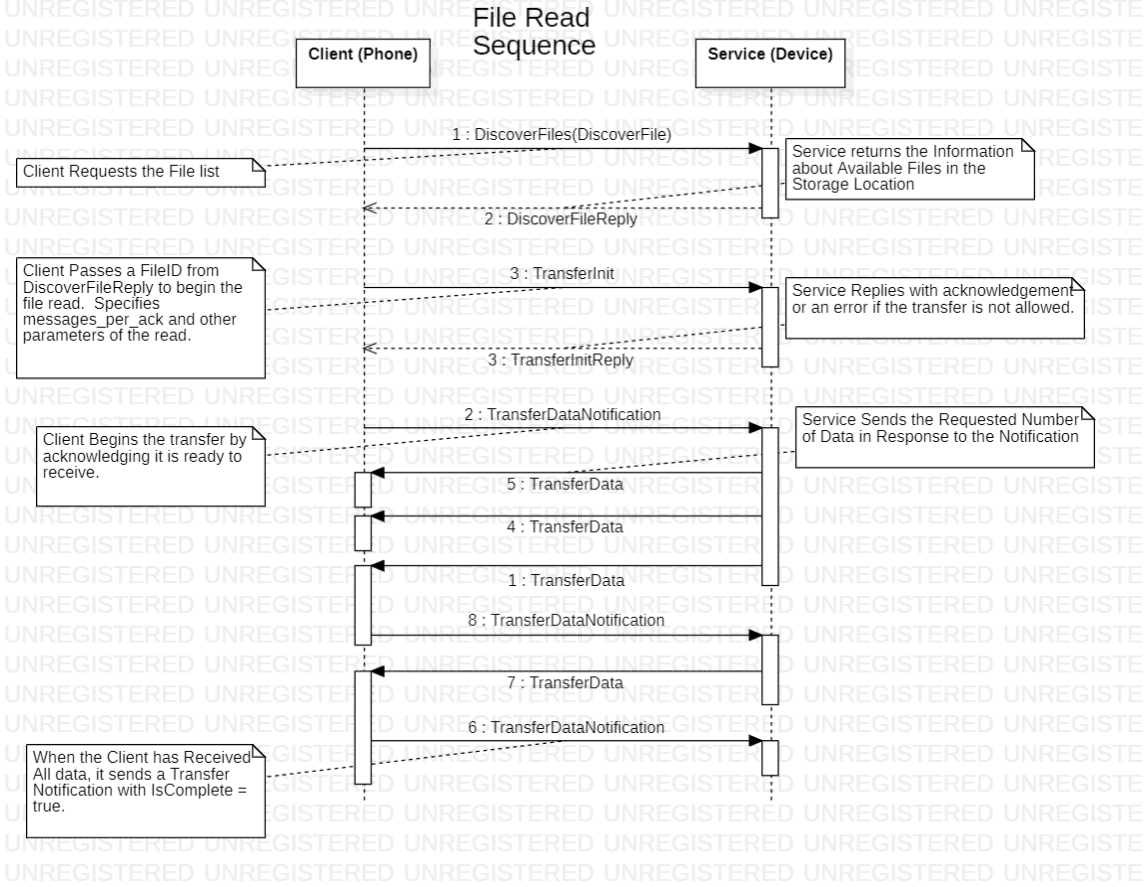
(Phone sends) Transfer Data Notification (SW CTS):

- Phone should be able to timeout.

(Device Responds) (multiple) Transfer Data messages.

- device sends N messages before waiting for an ACK.
- Phone should be able to timeout.

(phone sends) Final Transfer Data Notification, with “is_complete” : true



Write File Sequence

After the discover sequence, a file is written using a sequence of transactions. All files are binary data.

(Phone sends) Transfer Init

- File ID, read/write, size, offset,
- Messages per ACK
- Transfer_id

(device responds) Transfer Init Reply (error if not allowed)

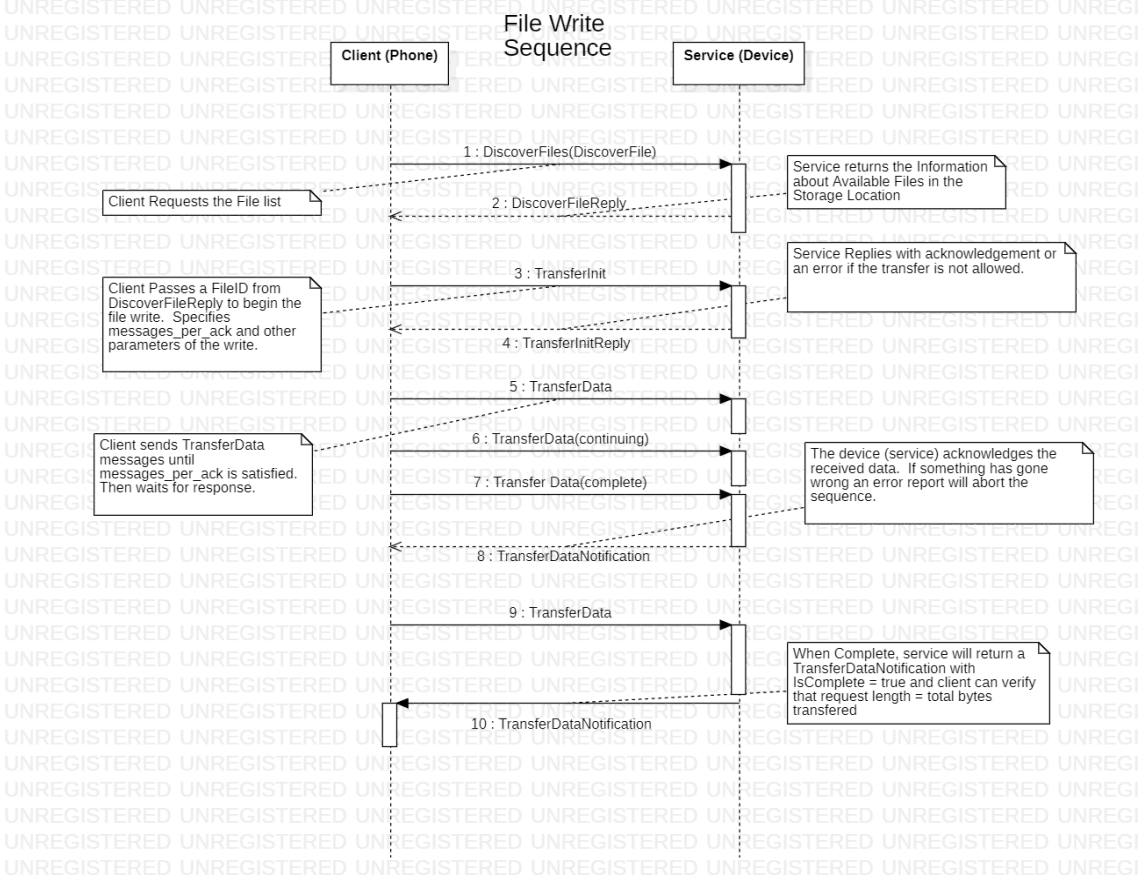
Repeat:

(Phone sends) (multiple) Transfer Data messages.

- phone waits for an ACK after N messages.
- Phone must be able to timeout if ACK is not received.

(device sends) Transfer Data Notification (SW CTS):

- OK, is_complete



Error Handling

As coded in V1, error messages are produced only in response to a prompt. This avoids the complications of unsolicited error messages at the client. It's conceivable that the error report (using cr_report_error()) could be an asynchronous notification like the CLI, but this is not done today. To enforce this, all of the functions called by cr_process() must behave in one of three ways:

1) Returning zero, the encoded buffer is populated and ready to be sent.
2) Returning CR_ERROR_NO_DATA indicates that there is no encoded data to be sent.
3) Other non-zero error returns indicate that cr_report_error() has been called to populate the encoded buffer with an error message.

Hence there can be at most one error report per call to cr_process(). cr_process() and all of its children are required to obey this rule of reporting an error. A clause in cr_process() checks this and informs the user if an error is returned without reporting.

Errors incurred in other context must be reported to the command line. The LOG_ERROR() macro exists for this purpose.

This remains to be tuned, but we consider it critical that there be a convenient and robust way to ensure that errors seen on the embedded server are visible on the remote client.

Remote CLI

Connecting the remote client to the CLI of the device is very valuable in development. It's particularly useful to access features controlled by the CLI. The "lm" command is an example here, changing the log mask. CLI messages are an example of data produced by the server without prompt by the client. This is handled in the V1 demo using an independent encoding chain and a separate socket client. It remains an open question whether this inefficient use of memory can be handled better with a BLE system. As is, there is a cost in memory allocation to using the remote CLI.

The remote CLI can be enabled or disabled using the "ENABLE_REMOTE_CLI" macro in reach-server.h.

Endpoints

The top level Reach message structure includes an endpoint entry. As this top level structure is for routing, we should use this to route messages to an endpoint. The device info structure is intended to include a map listing the endpoints that are available here. Endpoint zero contains this map.