ZCL/IP is an implementation of the Zigbee Alliance Dotdot specification, a generalization of all of the application layer functionality developed both in Zigbee PRO and the Zigbee Cluster Library (ZCL) for use over a variety of transports. Using ZCL/IP, Silicon Labs has developed a Thread application layer, which is provided with the Silicon Labs Thread stack. This document provides details on working with ZCL/IP.

For background information about Dotdot and ZCL/IP for Thread, see UG103.11: Thread Fundamentals.
1. Introduction

Zigbee Cluster Library over IP (ZCL/IP) is a transport-agnostic application networking layer protocol, developed and certified by the Zigbee Alliance as part of Dotdot, a more generalized application layer specification. ZCL/IP’s purpose is to establish a common protocol layer so that devices on different communication mediums can interact with each other seamlessly. The protocol is still under development, but making significant progress toward release to the public.

ZCL/IP is based on the Zigbee Cluster Library (ZCL), which is a data model of behaviors and states that devices take on. In the ZCL, these behaviors are referred to as commands and these states are referred to as attributes. For more information about the ZCL, see UG103.02: Zigbee Fundamentals.

ZCL/IP takes the commands and attributes described in the ZCL and defines an over-the-air protocol to use to exchange this information between devices. ZCL/IP uses the Constrained Application Protocol (CoAP) as its transport layer. In ZCL/IP messages, the CoAP protocol data unit contains information about the ZCL command or attributes to which the message pertains, and the service data unit contains information further describing the command or attributes. CoAP is very similar to HTTP. CoAP operates under a request/response communication scheme, and each message contains a code that specifies a request action (GET, PUT, POST, DELETE, etc.), response success (2.01 Created, 2.04 Changed, 2.05 Content, etc.), or response failure (4.00 Bad Request, 4.04 Not Found, 5.00 Internal Server Error). For more information about CoAP, see RFC 7252.

CoAP can theoretically run over any transport layer. Therefore, ZCL/IP theoretically runs over any transport, network, and physical layer. The Silicon Labs ZCL/IP implementation for Thread uses UDP as a transport layer, as this functionality is already included in the Silicon Labs Thread stack. For more information about Thread and the Silicon Labs Thread stack, see UG103.11: Thread Fundamentals.
2. Getting Started

2.1 Using ZCL/IP Example Applications

Whether you have had previous experience with Zigbee (perhaps through working with the Silicon Labs EmberZNet stack) or are completely new to Zigbee, the best place to get started with ZCL/IP is through the SL-Thread ZCL/IP examples applications in Simplicity Studio. A number of ZCL/IP examples are included in the SL-Thread release, but the best starting places are the Light and Switch example applications. These example applications both bring up communications on the same Thread network, and can then be controlled through development board buttons. The button interface for these example applications is explained in detail in the sample applications description, shown on the General tab of Simplicity Studio’s IDE, also known as AppBuilder. While using the buttons to send ZCL/IP messages between the Light and Switch sample apps, Silicon Labs recommends you have a Network Analyzer packet capture running, as this is an excellent way to better understand the ZCL/IP messaging format. This messaging format will be new to users with and without previous experience with Zigbee. The Light and Switch sample apps demonstrate many different kinds of ZCL/IP messages, such as, but not limited to, commands, notifications (reporting), and EZ-Mode. Implementing a Light/Switch simple network and using Network Analyzer are described in detail in QSG113: Getting Started with Silicon Labs Thread.

You can use the widgets provided in the AppBuilder ZCL over IP tab to configure the application by adding and removing endpoints, device types, clusters, commands, and attributes. Adding and removing information on the ZCL over IP tab in AppBuilder will result in changes to the code generated by AppBuilder, and therefore different configuration of the ZCL/IP application.

2.2 Using Existing EmberZNet ZCL Applications

You may wish to migrate some of the functionality from existing ZCL applications running over the EmberZNet PRO stack to ZCL/IP applications running over the Silicon Labs Thread stack. This functionality may include cluster implementations, special ZCL logic, and ZCL configuration. As the ZCL/IP protocol runs over a different stack, cluster implementations have been rewritten to support the Silicon Labs Thread Stack. Special ZCL logic from previous applications can certainly be reused in ZCL/IP applications. You can add source files to an AppBuilder application through the Additional Files table found in the Other tab. You may be able to reuse generic ZCL logic from previous application since the ZCL data model has stayed the same for ZCL/IP. Since AppBuilder uses the same user interface for both the Silicon Labs EmberZNet stack and the Silicon Labs Thread stack, you should be able to copy a previous application’s ZCL configuration by choosing the same options in the ZCL over IP tab of AppBuilder.
3. Configuration

3.1 CoAP

The Silicon Labs ZCL/IP implementation relies on the Constrained Application Protocol or CoAP for its transport layer. The Constrained Application Protocol is defined in RFC 7252. Silicon Labs includes a CoAP implementation along with the Silicon Labs Thread Stack. The CoAP implementation used for ZCL/IP is located at:

```
<stack install location>/app/coap
```

The CoAP implementation provides APIs for reliably sending and receiving messages over the Thread network.

**Sending a message using CoAP**

Messages can be sent over CoAP using the API `emberCoapSend`. Helper APIs named `emberCoapGet`, `emberCoapPut`, `emberCoapPost`, and `emberCoapDelete` also end up calling `emberCoapSend`.

**Receiving a message using CoAP**

All CoAP messages meant for the application are passed to the function `emberCoapRequestHandler`. The Silicon Labs Thread Application Framework has a plugin named CoAP Dispatch that multiplexes the calls to `emberCoapRequestHandler` based on the method with which the CoAP message was sent and the URI to which the CoAP message was sent.

For more information about the Silicon Labs Thread Stack CoAP API, see the CoAP module in the Silicon Labs Thread Stack API Reference Guide.

3.2 ZCL

The Zigbee Cluster Library is configured through the ZCL over IP tab in AppBuilder. If you are familiar with EmberZNet, configuration is the same as for any Zigbee application.
All of the code for Silicon Labs ZCL/IP implementation is shipped with the Thread stack and is located at:

<stack install location>/app/thread/plugin/zcl

This directory includes an ever-growing list of plugins that comprise the Silicon Labs ZCL/IP application layer implementation. The most important of these and probably the best place to begin looking into the code is the ZCL Core plugin.

The ZCL Core plugin provides the foundation for the ZCL/IP implementation including but not limited to such basic functionality as:

- Attribute management
- Binding management
- Command handling and dispatching
- Endpoint management
- Group management
- Reporting configuration management
- Notification handling and dispatching
- Device discovery
- Application provisioning
- General ZCL/IP utilities

For more information about the ZCL Core plugin API, see the ZCLIP module in the Silicon Labs Thread Stack API Guide.
4. Using DTLS with ZCL/IP

As the Silicon Labs ZCL/IP implementation uses CoAP as the underlying mechanism for messaging, it can also use Secure CoAP functionality in order to send data securely between two nodes.

4.1 Transport Layer Security (TLS) Methodologies

Silicon Labs has implemented Secure CoAP functionality on two different TLS (transport layer security) code bases: the Silicon Labs custom implementation and ARM’s mbed TLS.

Silicon Labs Thread and ZCL/IP communications security is, by default, provided through the Silicon Labs implementation of TLS. It is a stable and well-tested protocol that has been in use since 2010.

This implementation provides the following features required by Thread and ZCL/IP while keeping application image size small enough to flash onto devices with less than 256 kB data storage, including the EM3x and EFR3xG1 families.

- An SSL / TLS (3.3) implementation
- AES crypto support
- ECC support for ECJPAKE, and ECDHE, ECDSA support for secp256r1 (standardized 256-bits NIST) curves
- SHA-256 hash algorithm
- Pre-compiled X.509 certificates
- ECC private and public keys: PKCS#8 unencrypted private key and PKCS#8 public key modes.

For more information on Thread and Dotdot security requirements, see UG103.11: Application Development Fundamentals: Thread, and the zcl/ip base device specification.

Beginning with Silicon Labs Thread release 2.3, users who are developing for any device with >256 kB data storage, such as the EFR32xG12, have the option of using ARM mbed TLS for communications security. ARM mbed TLS is a widely-used and highly configurable implementation that supports additional features, including:

- SSL version 3
- Variety of key exchange methods (RSA, DHE-RSA, ECDH-RSA, ECDH-RSA, PSK, DHE-PSK, and others)
- Variety of cryptographic algorithms (AES, 3DES, DES, XTEA, and others)
- Various modes of operations (CBC, EBC, CFB, and others)
- Various hash algorithms (MD2, MD4, SHA-1, SHA-256, and others)
- ECC support including ECDHE, ECDSA and ECJPAKE, among others
- NIST standardized CTR_DRBG and HMAC_DRBG random number generators
- Supports a variety of underlying technologies – X.509 certificates; RSA and ECC private and public key reading
For more information on ARM mbed TLS, see https://tls.mbed.org/core-features and https://tls.mbed.org/standard-compliance. For more information on using mbed TLS when implementing a secure CoAP solution, see AN1083: Creating and Using a Secure CoAP Connection with ARM’s mbed TLS.

Plugins in the Stack group on the Plugins tab control whether Silicon Labs TLS or ARM mbed TLS is used in your application. The stack group is shown in the following figure.

For Silicon Labs TLS, select the following two precompiled libraries. This is the default configuration for example applications:

- Thread Stack Library with legacy TLS support
- mbed TLS Cryptographic Library: Provides hardware crypto acceleration support, use only with the thread-stack library.

For ARM mbed TLS in an implementation that provides all functionality required by ZCL/IP, select the following two precompiled libraries:

- Thread Stack Library with mbedTLS support
- mbed TLS Library
For ARM mbed TLS in a custom stack implementation that you can build, select the following two libraries:

- Thread Stack Library with mbed TLS support
- mbed TLS Custom Library: Includes a configurable option to use a custom mbed TLS header. By default, this option takes a custom header that is configured for minimal stack (commissioning) support. The header can be overwritten using one with additional options.
4.2 Example DTLS Usage in ZCL IP

The ZCL Core DTLS Manager CLI plugin, which provides CLI access to the secure CoAP functionality, is available in both the Light and Switch example applications.

When the ZCL Core plugin is enabled and a DTLS session is established, the ZCL/IP messaging API sends messages over the secure session. When a secure message is received, it is parsed through the emberProcessCoap function in app/thread/plugin/zcl/zcl-core/zcl-core-dtls-manager.c

To see this in operation, create the Light and Switch examples, start a network, and observe interactions in Network Analyzer, as described in QSG113: Getting Started with Silicon Labs Thread. Notice that, when toggling the light from the switch, Network Analyzer shows regular CoAP messages.

```
CoAP con 0.02 POST /zcl/e/1/c6/n [1/2]
CoAP con 0.02 POST /zcl/e/1/c6/n [2/2]
```

To set up and observe a DTLS connection:

1. Get the IP Address of the light using:

```
light> info
```

```
network status: 0x03
emu64: >000B57FFFE25FFA3
network id: precommissioned
node type: 0x02
extended pan id: >4F8EC75FB4E1EFC6
pan id: 0x8394
channel: 24
radio tx power: 4 dBm
ula prefix: fd01::/64
local ip 0: fd01::e48a:723f:8730:140d
local ip 1: fe80::70d7:25e3:b9d0:4f6a
```

2. If you are using a pre-shared key (PSK) for the secure connection, you need to configure the keys you plan to use on both sides of the connection. To do this, issue `zcl dtls-manager psk "<remoteIP>" "<secretKey>" "<identityHint>"` on the SoC. Then a connection can be made using the key `secretKey` from a device at `remoteIP`. The `identityHint` is a feature based on MBEDTLS's implementation for having multiple PSKs and selecting based on an identity. This is currently unsupported.

```
Authentication parameters used in x509 will be compiled in or written to flash at manufacturing time. See section 4.3 DTLS Authentication Parameters for more information.
```

3. Open the DTLS connection using:

```
switch> zcl dtls-manager open <securityMode> <ip> <port>
```

```
securityMode is a set of flags comprising:
0x01 – x509 Certificates
0x02 – PSK
0x04 – Preshared Key
```

4. DTLS Session established callbacks will fire on both nodes

```
light> secure session available: 1 (server)
switch> secure session available: 1 (client)
```

5. Use the application securely: When toggling the light from the switch, Network Analyzer shows Secured messages.

```
DTLS:application_data [1/2]
DTLS:application_data [2/2]
```
4.3 DTLS Authentication Parameters

Regardless of whether you are using Silicon Labs TLS or ARM mbed TLS, the DTLS Authentication Parameters (dtls-auth-params) plugin must be selected. This plugin provides a stub implementation of the certificates required for authentication, as well as “real” certificates for the ZCL examples. Refer to the following files for the code used to generate the stub and demo certificates:

- app/thread/plugin/dtls-auth-params/stub-certificates.c: Stub / empty certificate definitions
- app/thread/plugin/dtls-auth-params/sample-dotdot-certificates.c: Sample X.509 certificates being used to demo ZCL apps.

The following examples show how to create custom X.509 certificates for use with the dtls-auth-params plugin. The examples are intended for a Linux or other Unix (Mac OS X, etc.) environment that has the OpenSSL libraries installed. If you are using a different environment, consult documentation for how to create .pem files.

4.3.1 Creating Your Own Certificate Authority

1. Generate a Certificate Authority (CA) private key using an Elliptical Curve (EC).

```bash
openssl ecparam -genkey -name secp256r1 -out ca-key.pem
```

2. Create a CA certificate using the above private key.

```bash
openssl req -x509 -new -SHA256 -nodes -key ca-key.pem -days 3650 -out ca-cert.pem
```

3. Create a Certificate Signing Request (CSR) using the private key.

```bash
openssl req -new -SHA256 -key ca-key.pem -nodes -out ca-csr.pem
```

4. Verify the details in the CSR.

```bash
openssl req -in ca-csr.pem -noout -text
```

5. Generate an unencrypted pkcs8 format key for the DTLS plugin by executing this command:

```bash
openssl pkcs8 -topk8 -nocrypt -in ca-key.pem -out ca-pkcs8-key.pem
```
4.3.2 Creating Your Own Device Certificate

To create your own device certificate (in the format that Dotdot applications require), use the Privacy-Enhanced Mail (PEM) files generated in section 4.3.1 Creating Your Own Certificate Authority, or from another CA authority, and execute this command:

```bash
openssl x509 -req -SHA256 -days 3650 -in ca-csr.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial -out device-cert.pem
```

4.3.3 Converting PEM Files into C Files for Use in a DTLS Application

The last step is to convert the PEM files generated in the two previous steps into a .c file that the DTLS Authentication Parameters (dtls-auth-params) plugin can accept as an input.

1. Save all the .pem files generated in the previous two procedures to your hard drive.
2. Change to the directory `app/thread/plugin/dtls-auth-params`
3. Execute this command.

   ```bash
   ./translate-certificates.sh ca-cert.pem ca-key.pem ca-pkcs8-key.pem device-cert.pem
   ```

   This creates a .c file named translated-certificates.c.

4. (Optional) Rename translated-certificates.c.
5. Use translated-certificates.c as an option to the DTLS Authentication Parameters (dtls-auth-params) plugin.
5. Using the OTA Upgrade Solution in ZCL/IP

The Ddotdot OTA (Over-The-Air) Upgrade (OTA) solution is based on the analogous solution in Zigbee PRO. The Ddotdot OTA solution takes the concepts specific to Zigbee PRO from the OTA Upgrading cluster (as specified in chapter 11 of the Zigbee Cluster Library (ZCL) specification), and generalizes them so that they can be used in the Ddotdot transport-agnostic application layer. In addition to this information, please see AN716: Image Builder Instructions and AN728: Over-the-Air Bootloader Server and Client Setup for more information about the Zigbee PRO OTA solution. For information on manufacturing certificates see AN1163: Using Manufacturing Certificates on ZCL/IP Devices.

5.1 OTA Functionality in Example Applications

The Silicon Labs Thread 2.3 and higher release contains three example applications that demonstrate Ddotdot OTA functionality. The Light and Switch examples function as OTA Bootload Client devices, and the Border Router Management Application functions as an OTA Bootload Server device. The Ddotdot OTA functionality can be configured through the OTA Bootload Client (shown below) and OTA Bootload Server plugins in the ZCL plugins group.
In addition to these plugins, the OTA Bootload Client Policy (shown below) and OTA Bootload Server Policy plugins provide a helpful abstraction for configuring an application to run as an OTA device.

There are three types of server detection methods: Discover, Static IP, and DNS. Each method uses the subsequent parameters in different ways.

- **Discover**
  
  This method tries to discover an update server that has announced itself on the mesh. If multiple servers are on the mesh, the `IPV6 Address` parameter can be set to that of the desired server. This acts as a filter to discover only the server that matches the given address. The same function can be performed with the `Port`, `UID`, and `Endpoint` parameters. Note, this is not a mask value, but rather an explicit match. A wildcard (all 0xFF) will match anything. The `OTA Server Name` is unused.

- **Static IP**
  
  All server parameters (with the exception of `OTA Server Name`) are used exactly as given to communicate with an OTA server. No parameter may be a wildcard (all 0xFF).

- **DNS**
  
  A DNS server is necessary for this setting. It is typically implemented via a Border Router that is on the mesh. The Border Router declares itself as having Network Discovery services. Using the `OTA Server Name` field, a query for the IPv6 address is initiated. The `IPV6 Address` field is ignored. The `Port`, `UID`, and `Endpoint` parameters are used as they appear in AppBuilder. These parameters cannot be wildcards (all 0xFF).

Depending on the method selected, the device will seek out the update server. An on-mesh upgrade server example is located in the Light (HOST) sample application. For upgrades in the cloud, the zclip Javascript sample server is offered. This server may be accessed through a Border Router.
5.2 Using the Dotdot OTA Image Builder Utility

The Silicon Labs Thread 2.3 release contains a simple utility to generate an OTA file for a provided image file (such as an EBL or GBL file). This utility is similar to, but simpler than, the Image Builder utility provided in the EmberZNet SDK.

OTA files are a wrapper around binary data, such as upgrade images. The OTA file wrapper enables the data to be transferred securely over the air from an OTA Bootload Server device to an OTA Bootload Client device. An OTA file starts with a header that describes the data contained in the file. The header contains fields such as, but not limited to, the manufacturer code, the type, and the version describing the binary data. The primary purpose of the Image Builder utility is to wrap a provided image file with this OTA header so that the file can then be transferred over the air.

The interface for the Silicon Labs Thread Image Builder utility has five required command line arguments:

- The input image file name
- The output OTA file name
- The manufacturer code of the input image file (16-bit)
- The type of the input image file (16-bit)
- The version of the input image file (32 bit).

Here is an example invocation of the utility.

```
image-builder-windows.exe light.gbl 1234-ABCD-10001000-light.ota 0x1234 0xABCD 0x10001000
```

The preceding command generates an OTA file named 1234-ABCD-10001000-light.ota, which contains the image file light.gbl, and has a manufacturer code of 0x1234, an image type of 0xABCD, and a file version of 0x10001000. This 1234-ABCD-10001000-light.ota file can then be added to an OTA Bootload Server and downloaded to an OTA Bootload Client for upgrade.

The Image Builder executable is located in `<studio installation>/developer/sdks/gecko_sdk_suite/v1.1/protocol/thread_n.n/tool`.

Note: The Image Builder utility found in a parallel location in the EmberZNet release should not be used to create Dotdot OTA files for use in ZCL/IP.

5.3 Comparison with Zigbee PRO OTA

The basic structure of the OTA file format has stayed the same. For example, an OTA file can still be uniquely identified by the combination of a manufacturer code, a file type, and a file version. The only changes made to the OTA file format are simple generalizations. For example, instead of specifying which version of the Zigbee networking stack the OTA file applies to, the file specifies the type of general communication stack for which it is meant. In addition, the concept of an IEEE address has been replaced with a Unique Identifier (UID). A UID is now used anywhere that an IEEE address was used in the Zigbee PRO OTA solution. Using a UID provides more authenticity and applicability to a general communication stack. Finally, it is a mandate that these Dotdot OTA files are secured using IP security credentials, to ensure a secure transport of OTA files.

There are two main differences between the Zigbee PRO and Dotdot OTA designs. The first main difference is the image transfer mechanism. In Dotdot OTA, the communication protocol with which the client and server devices transfer the image is left up to the application. A manufacturer is free to decide the method that they would prefer to use when sending an OTA file to one or more client devices. In the current Dotdot OTA solution, the transfer mechanism of choice is Constrained Application Protocol (CoAP) block transfer, as specified in RFC7959. CoAP functionality is already required in Dotdot implementations, so this method of file transfer should fit in easily for applications, while still maintaining manageable code size. Other examples of image transfer mechanisms are multicast block transfer and server “push.”

The second main difference between Zigbee PRO and Dotdot OTA designs is server discovery. While it is possible that a client device exists on the same mesh network as its designated upgrade server, in reality, many manufacturer will wish to store their upgrade images in an off-mesh location, perhaps in the cloud. The Dotdot OTA solution provides the ability for client devices to communicate with and download an image from a server device that exists off-mesh. A manufacturer can write the domain name of a server device to a client device, and the client device can then use Domain Name Service (DNS) to resolve the domain name into an IPV6 address to use to communicate with the server device. This feature enables manufacturers to cut out a large step in their upgrade process, solving many issues raised against the Zigbee PRO OTA solution.
6. Discovering, Controlling, and Monitoring Thread Devices with zclip.js

6.1 JavaScript Tools Available for Demonstration and Development

Zclip.js is a Java Script implementation of the ZCL-over-IP (ZCL/IP) application layer defined by the Zigbee Alliance. It is available at https://github.com/SiliconLabs/zclip.js and provides libraries for creating custom ZCL/IP applications.

Silicon Labs Thread Border Router Reference Design ships with two sample zclip.js-based applications to aid in demonstration and development: zclip-cli and zclip-rd server.

- **zclip-cli** is a command line interface that supports ZCL/IP device discovery and control. It is available at https://github.com/SiliconLabs/zclip-cli and may be installed on any computer with IPv6 connectivity to the Border Router.

- **zclip-rd server** is a java script implementation of a ZCL/IP Resource Directory (RD). An RD provides both a database of ZCL/IP devices currently on the Thread network as well as a server to handle device database registration and queries. It facilitates a number of device management and control tasks including lookup of sleepy devices and remote interaction. zclip-rd should be installed on a border router. Source code for this tool is available at https://github.com/SiliconLabs/zclip-rd

The following sections explain how to use the preinstalled sample applications. See the documentation on github for an explanation of the dependencies between zclip.js, zclip-rd, and zclip-cli.

6.2 Device Discovery

Two methods can be used to discover devices on a Thread network: Resource-Directory Lookup and Multicast Discovery. Both are supported by zclip-cli.

6.2.1 Device Discovery Using the RD

*Resource Directory Lookup* requires that an RD server be present on the border router. Furthermore, devices must register with the RD. Specifically:

1. Devices on the Thread network must have the RD client plugin enabled and must actively register with the RD.
2. Remote RD clients must know the IP address of the border router in order to reach the RD from off-mesh.

**Registering with the resource directory**

Using Simplicity Studio, connect to the end device and launch a debugging console. Issue the following command:

```
> zcl resource-directory register "<rdIp>" <rdPort>
```

If registration is successful, the following will be returned:

```
> Registering with resource directory
> Registration Success
```

**Device lookup using the resource directory**

Establish an SSH connection with the border router following the instructions in [UG116: Developing Custom Border Router Applications](https://github.com/SiliconLabs/zclip-rd). To discover devices using the RD, issue the following command:

```
pi@br-51B0:~ $ zcl lookup --rdIp <ipv6-address> --rdPort 5689 --page 0 --count 4
```

The paging parameters (page and count) are required when the lookup result exceeds the maximum payload size for a CoAP packet. In practice, the payload handles a count of 4 without issue.

See https://github.com/SiliconLabs/zclip-cli for more usage examples, such as querying for only specific clusters.
6.2.2 Device Discovery Using Multicast

When no RD is present, discovery can still be achieved by multi-casting a discovery message across the Thread network. However, not all devices will respond.

1. Devices that have registered with an RD may not respond to multicast discovery messages by design.
2. Sleepy devices are unlikely to be awake to hear multicast discoveries at the time the message is sent.

Zclip-cli reverts to multi-cast discovery if no resource directory is specified. For example, devices that support the onOff cluster will respond to the following multicast discovery:

```
pi@br-51B0:~ $ zcl lookup --cluster onOff
```

6.3 Device Interaction

Once a device has been discovered, it can be read or controlled using zclip-cli. The target device can be specified by its IPv6 address as follows:

```
zcl cmd <cluster> <command> --ip <ip> [arguments]
```

For example:

```
zcl cmd levelControl move --ip ::1 --moveMode 1 --rate 255
```

One disadvantage of using the ipv6 address in this way is that the address can change at any time. A preferred way to specify the target device is using its UID as follows:

```
zcl cmd <cluster> <command> --uid <uid> --rdIp <RDIP> --rdPort <RDPort> [arguments]
```

For example:

```
zcl cmd levelControl move --uid P7BU0eh27b2f5f0IC0GLL7uHum_DtmRVe73umRQCAeY --rdIp ::1 --rdPort 5689 --moveMode 1 --rate 255
```

The UID-specified method requires that we provide the location of a RD. This is because UIDs are not usable for global routing. Therefore, when you attempt to identify the target device using UID, zclip-cli actually performs two transactions behind the scenes:

1. Zclip-cli queries the RD in order to find the IPv6 address that matches the UID.
2. Zclip-cli then sends the message to the target using the IPv6 address.

The purpose of first querying the RD is that it should have the correct IPv6 address of the target, because all devices are supposed to update their registration with the RD any time their IPv6 address changes.

See https://github.com/SiliconLabs/zclip-cli for more usage examples, including attribute reads and binding.
Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice to the product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Without prior notification, Silicon Labs may update product firmware during the manufacturing process for security or reliability reasons. Such changes will not alter the specifications or the performance of the product. Silicon Labs shall have no liability for the consequences of use of the information supplied in this document. This document does not imply or expressly grant any license to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any FDA Class III devices, applications for which FDA premarket approval is required or Life Support Systems without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons. Silicon Labs disclaims all express and implied warranties and shall not be responsible or liable for any injuries or damages related to use of a Silicon Labs product in such unauthorized applications.

Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world’s most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, Gecko OS, Gecko OS Studio, ISOModern®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress®, Zentri, the Zentri logo and Zentri DMS, Z-Wave®, and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of the Wi-Fi Alliance. All other products or brand names mentioned herein are trademarks of their respective holders.

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com