Implementation Report: Advanced Load Up Command in CTA-2045-B

Revision History

Date	Version	Description	Author
12/9/2024	0.1	Initiate doc	M. Adham
12/10/2024	0.2	Added progress. Rheem WH ALU implementation.	M. Adham
12/18/2025	0.3	Progress on AOSmith WH. A different approach.	M. Adham
12/27/2025	0.4	Trying different ALU parameters	M. Adham
12/28/2025	0.5	Finalizing the ALU implementation	M. Adham

Initial Approach and Setup

We examined the CTA-2045-B specification to understand the Advanced Load Up functionality. The primary goal was to enable devices to store extra energy beyond their normal capacity. In this implementation, an 80-gallon Rheem water heater was used.

Step 1: Understanding Capability Bitmap

We first discovered that Advanced Load Up required setting bit 6 in the capability bitmap. This led us to:

1. Examine the capability bitmap structure in CEA2045MessageDeviceInfoResponse.h
2. Add getter methods for capability bits:

```
bool getCapabilityBit(unsigned char bitNum)
bool getExtendedCapabilityBit(unsigned char bitNum)
bool isAdvancedLoadUpEnabled()
```

Step 2: SetCapabilityBit Implementation

We implemented the SetCapabilityBit functionality through several files:

1. CEA2045MessageSetCapabilityBitRequest.h:
   - Defined message structure
   - Added proper bit manipulation for capability setting
2. SetCapabilityBit.cpp:
   - Implemented message construction
   - Set correct opCodes and length
   - Added proper checksum calculation

Step 3: Testing and Debugging

Initial testing revealed several issues:

1. Message Length Error:
   - Received response code 0x03 (length error)
   - Fixed by ensuring correct payload length (4 bytes)
2. Capability Bit Response:
   - Successfully set bit 6 (responseCode: 0x0)
   - But capability remained disabled in device info response

Step 4: Endianness Investigation

We discovered potential endianness issues when reading device information:

1. Original Working Code:

```
unsigned short getVendorID() {
   return be16toh(vendorId);  // Converts from big-endian to host
}
```

2. Our Implementation Initially:
   - Missing byte-order conversion
   - Raw bytes showing zeros instead of actual values
3. Solution:
   - Added proper byte-order conversion using be16toh()
   - Implemented getters with correct endianness handling
   - Fixed device info response parsing

Step 5: Device Info Message Format

We identified that device info requests needed specific formatting:

1. Working code:

```
new Intermediate(MessageCode::DEVICE_INFORMATION_REQUEST,
      INFO_REQ, INFO_REQ)  // Both opCodes set to 0x01
```

2. Important Constants:

```
#define INFO_REQ 0x01
#define DEVICE_INFO_RESPONSE_OP_CODE1 0x01
#define DEVICE_INFO_RESPONSE_OP_CODE2 0x81
```

Current Status and Next Steps

1. Successfully implemented message structures
2. Verified capability bit setting works (gets success response)
3. Need to resolve endianness issues for device info response
4. Need to verify proper parsing of capability bitmap

Key Files Modified

1. CEA2045MessageDeviceInfoResponse.h
2. CEA2045MessageSetCapabilityBitRequest.h

3. SetCapabilityBit.cpp
4. ProcessMessageUCM.cpp
5. ConvertEnums.h

Next Steps

1. Implement proper endianness handling for device info response
2. Add comprehensive debug logging for message parsing
3. Verify capability bitmap handling in both directions
4. Complete end-to-end testing of Advanced Load Up Functionality

Progress 01/16/2025

The DeviceInfo does not print the bitmap capability by default. I was able to print the bitmap capability by adjusting the following files:

main.cpp

In the switch function, I added

```
cout<<"d- Device Info\n";
```

and

```
case 'd':
                device->intermediateGetDeviceInformation().get();
```

ProcessMessageUCM.cpp

The DeviceInfo message is processed in `ProcessMessageUCM::processIntermediateMessage()` under the `IntermediateTypeCode::INFO_RESPONSE` case. So, I modified the ProcessMessageUCM.cpp by adding the following lines:

```
case IntermediateTypeCode::INFO_RESPONSE:
        {
            cea2045DeviceInfoResponse *infoResponse = (cea2045DeviceInfoResponse
*)message;


            m_ucm->processDeviceInfoResponse(infoResponse);
```

```
std::cout << "Device Capabilities:" << std::endl;


// Combine the 4 bytes into a 32-bit value for easier bit testing

uint32_t capabilities = (infoResponse->capability[0] << 24) |

                        (infoResponse->capability[1] << 16) |

                        (infoResponse->capability[2] << 8) |

                        infoResponse->capability[3];


// Check each defined capability bit

if (capabilities & (1 << 0))

    std::cout << "- Cycling supported" << std::endl;

if (capabilities & (1 << 1))

    std::cout << "- Tier mode supported" << std::endl;

if (capabilities & (1 << 2))

    std::cout << "- Price mode supported" << std::endl;

if (capabilities & (1 << 3))

    std::cout << "- Temperature Offset supported" << std::endl;

if (capabilities & (1 << 4))

    std::cout << "- Continuously variable power" << std::endl;

if (capabilities & (1 << 5))

    std::cout << "- Discretely variable power" << std::endl;

if (capabilities & (1 << 6))

    std::cout << "- Advanced Load Up supported" << std::endl;

if (capabilities & (1 << 7))

    std::cout << "- Price Stream supported" << std::endl;
```

```
        if (capabilities & (1 << 8))

            std::cout << "- SGD Efficiency Level supported" << std::endl;


        // Print raw capability bytes for debugging

        std::cout << "Raw capability bytes: "

                << std::hex

                << (int)infoResponse->capability[0] << " "

                << (int)infoResponse->capability[1] << " "

                << (int)infoResponse->capability[2] << " "

                << (int)infoResponse->capability[3]

                << std::dec << std::endl;


        linkLayer->sendLinkLayerAck();


        break;

    }
```

Now, when we send a DeviceInfo message (d), we get the following response:

```
Unset
d

2025-01-16 17:39:23,065 INFO  [default] ack received: 15

2025-01-16 17:39:23,195 INFO  [default] message type supported
received: 2
```

```
2025-01-16 17:39:23,195 INFO  [default] device info response
received

2025-01-16 17:39:23,196 INFO  [default]      device type: 3

2025-01-16 17:39:23,196 INFO  [default]      vendor ID: 280

2025-01-16 17:39:23,196 INFO  [default]   firmware date: 2040-149-0

Device Capabilities:

- Advanced Load Up supported

- Price Stream supported

- SGD Efficiency Level supported

Raw capability bytes: 0 0 1 c0
```

The highlighted output is the bitmap capability. This output can be broken down into the following (as per the CTA-2045-B documentation):

```
Unset
bytes[0] = 0x00

bytes[1] = 0x00

bytes[2] = 0x01 <-- Bit 6 (Advanced load up = 1, meaning it is
supported)

bytes[3] = 0xC0 (binary: 11000000) <-- Bit 7 and 8 = 1, meaning that
price stream and SGD efficiency level are also supported.
```

The process is breaking down at the link layer. The ReceiveBuffer does not print anything when we send an advanced load-up, which means something is wrong.

Unset
```
CEA2045DeviceUCM::intermediateSetAdvancedLoadUp called
Creating Advanced Load Up message...
Message structure size: 24
Header size: 4
Setting message fields:
Duration: 60
Value: 5
Units: 2
SuggestedEfficiency: 255
EventID: 0
Calculating length:
Total size: 24
Header size: 4
Payload size: 20
Message length set to: 18
Queueing message with code: 23

LinkLayerCommImpl::send - Full message dump:
08 02 00 12 0c 00 00 3c
00 05 02 ff 00 00 00 00
00 00 00 00 00 00 53 96

We are inside the if statement
ReceiveBuffer received 2 bytes: 6 0
2025-01-17 15:13:56,884 INFO  [default] ack received: 23
We are inside the if statement
ReceiveBuffer received 9 bytes: 8 2 0 3 c 80
2025-01-17 15:13:56,988 INFO  [default] message type supported received: 2
LinkLayerCommImpl::send - Full message dump:
15 07
```

The 07 in the link layer response means the message is not supported. Check CTA-2045 doc page 16.

Progress Summary

a. Link Layer:

1. When we send an Advanced Load-Up message, we get a link layer Ack:

```
Unset
        LinkLayerCommImpl::send - Full message dump: 08 02 00 12 0c 00 00 3c ...
        (Advanced Load Up message)


        ReceiveBuffer received 2 bytes: 06 00 (This is Link Layer ACK)
```

2. This acknowledgment indicates that the WH is receiving and acknowledging the message and that the message is in the correct format.

b. Application Layer:

1. In the application layer, we get the actual response from the water heater.

```
Unset
ReceiveBuffer received 9 bytes: 08 02 00 03 0c 80 02 97 22 (Application layer
response)
```

Response Code	**Interpretations**
08 02	Intermediate message type
00 03	Message Length = 3
0c 80	Advanced Load-Up Response
02	Response Code = Bad Value
97 22	Checksum

2. We, in return, send an ack (Acknowledging receiving the message)

So, the water heater is receiving the command correctly (Link Layer Ack) but responding with the inability to perform the command (application layer - Nak).

Next Steps

1. Try the current implementation on another water heater.
2. If it is not working, debug the progress and see what next steps can help us move forward.

Progress 01/27/2025

General Notes

We noticed a problem when we sent the advanced load-up message with a set of padded zeros. This structure led to an unanticipated increase in the message length (10 bytes). Removing these zeros at the end fixed the issue, and the advanced load-up is now working. The next steps are to show how the AOSmith water heater behaves when an advanced load-up command is sent.

Progress 01/28/2025

Logs

Note the difference between the advanced energytake (1725) and the basic energytake (225)

```
Unset
Tue Jan 28 11:31:19 2025, 0, 181113, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:32:19 2025, 0, 181188, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:33:20 2025, 0, 181263, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:34:20 2025, 0, 181338, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:35:21 2025, 0, 181413, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:36:21 2025, 0, 181488, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1875, 0, 0, 3
Tue Jan 28 11:37:22 2025, 0, 181563, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1800, 0, 0, 3
Tue Jan 28 11:38:22 2025, 0, 181638, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1800, 0, 0, 3
Tue Jan 28 11:39:23 2025, 0, 181713, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1800, 0, 0, 3
Tue Jan 28 11:40:23 2025, 0, 181788, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1800, 0, 0, 3
Tue Jan 28 11:41:24 2025, 0, 181863, 4500, 6, 5250, 0, 7, 225, 0, 10, 7125, 0,
11, 1800, 0, 0, 3
```

Response Codes

```
Unset
Current buffer contents (9 total bytes):
08 02 00 03 0c 80 00 9b 20
ProcessMessageUCM::processIntermediateMessage:
  opCode1: 0xc  opCode2: 0x80
Advanced Load Up Response - Code: 0x0
Success
LinkLayerCommImpl::send - Full message dump:
06 00
```

As per the CTA-2045-B, the response codes shown below can be interpreted as follows:

Response Code	**Interpretations**
08 02	Intermediate message type
00 03	Message Length = 3
0c 80	Advanced Load-Up Response
00	Response Code = Success
9b 20	Checksum

Understanding CTA-2045-B Advanced Load-up Implementation

When the CTA-2045 program is compiled, the user may type “a,” executing the advanced load-up command. This execution goes through multiple files, including constructing the command message, decoding the responses from the link and application layers, and adding and removing the message from the queue. The following paragraph shows the message flow from the main function until it is executed.

1. `main.cpp`:
    a. User inputs 'a'
    b. Creates parameters (duration=60, value=5, units=0x02)
        i. These can be changed from the `main.cpp` file.
        ii. I tried several other values, and all of them worked.
        iii. Refer to page 79 in the CTA-2045-B doc.
    c. Calls `device->intermediateSetAdvancedLoadUp()`

2. `CEA2045DeviceUCM.cpp`:
    a. `intermediateSetAdvancedLoadUp()` is called
    b. Creates new `SetAdvancedLoadUp` message object
    c. Calls `queueRequest()` inherited from `CEA2045Device`

3. `SetAdvancedLoadUp.cpp`:
    a. Constructor builds message structure
    b. Sets message type, opcodes, values
    c. Sets length and calculates checksum

4. Back to `CEA2045Device.cpp`:
    a. `queueRequest()` adds the message to the queue
    b. `processNextRequest()` gets the message from queue
    c. Calls `m_linkLayer->sendRequest()` to send message

5. `LinkLayerCommImpl.cpp`:
    a. `sendRequest()` sends bytes over the serial interface
    b. Waits for response

6. When the response comes back:

a. `ProcessMessageUCM.cpp` handles the response
b. `processIntermediateMessage()` processes the Advanced Load Up response.
c. Reports success/failure