OCPP 2.Lite

OCPP for Resource-Constrained Devices

Version 2, July 2025

Table of Contents

Version History

Version	Date	Description
1	March 2025	First edition
2	July 2025	Added case study appendices for MicroOCPP and OpenOCPP

Management summary

This document addresses the challenges of implementing the Open Charge Point Protocol (OCPP) 2.0.1 on resource-constrained devices, such as microcontrollers used in lightweight electric vehicle charging stations. With limited RAM, ROM, and flash memory available, it is impractical to deploy the entire protocol. The focus is on achieving a minimal yet compliant implementation, prioritizing the "Core" profile of OCPP while omitting optional features that are not essential for basic functionality.

Key strategies include minimizing firmware size, reducing memory usage, and optimizing message handling. For example, read-only variables avoid the need for embedded databases, while smaller TLS buffer sizes help reduce memory demands. Real-world benchmarks from MicroOCPP demonstrate that a streamlined implementation can operate within 50 kB of RAM and 200 kB of ROM, making it viable for low-resource environments.

By tailoring OCPP to resource-limited devices, this approach ensures interoperability with existing systems while enabling cost-effective deployment of charging infrastructure. This balance between efficiency, scalability, and compliance is critical to supporting the growth of EV charging networks.

1. Introduction

The Open Charge Point Protocol (OCPP) is a communication protocol specifically designed for the exchange of information between electric vehicle (EV) charging stations and their management systems. It enables the authentication of EV drivers, manages charging sessions, and monitors the performance of charging stations.

Over the past 15 years, OCPP has evolved in tandem with the electric vehicle charging industry. Currently, the most widely adopted version of OCPP for resource-constrained devices is OCPP 1.6, which was published in 2015. This version encompasses essential functions for authorization and transaction handling, and it provides optional profiles for smart charging, security, and firmware management. Despite being a decade old, OCPP 1.6 remains actively supported by the Open Charge Alliance (OCA), which continues to manage errata, conduct testing events, and enhance the conformance testing tool and certification program. However, no new features are being added to OCPP 1.6; all new development efforts are focused on the OCPP 2.x series, which represents a significant advancement introduced in 2020.

Looking ahead, it is imperative that resource-constrained devices are able to benefit from the ongoing feature development in EV charging, implementing OCPP 2.x in a manner compatible with their hardware. For charging network operators, utilizing a single version of OCPP to manage their entire portfolio of charging stations—from fast chargers to basic wall boxes—will enhance operational efficiency. Furthermore, regulators can reference a single version of OCPP, thereby consolidating efforts and clarifying regulations. This paper will explore the implementation of OCPP 2.0.1 (or OCPP 2.x in general) in the most streamlined manner possible: OCPP 2.Lite.

The OCPP 2.0.1 specification comprises over 1,500 pages. In contrast, the OCPP 1.6 release is approximately 150 pages long, which might lead to the assumption that an OCPP 2.0.1 implementation would be ten times larger than that of OCPP 1.6. This is not necessarily accurate. This paper will demonstrate that a "light" OCPP 2.0.1 implementation, devoid of optional features, can be of comparable size or only marginally larger than OCPP 1.6.

Two minimal implementations, that are based upon the recommendation in this paper, have been developed: MicroOCPP [MicroOCPP] and OpenOCPP [OpenOCPP]. Their results suggest that the OCPP protocol stack of these implementations, that support both OCPP 1.6 and 2.0.1, will fit within 200 KB. The size of a full firmware image for a charging station will vary with the capabilities and features or a charging station, but is expected to remain below 2 MB.

These implementations are ready and available as open source. Their findings have been added as case studies in the appendices of this paper.

2. Flexibility for Certification Purposes: Profiles, Features, and Product Types

Globally, OCPP is utilized across a diverse range of charging stations, from basic to advanced models, and from low-power to high-power systems, including bidirectional charging stations. However, not all features outlined in the OCPP specification are universally required. As such, many features within OCPP are optional, enabling implementers to select functionalities that best align with their specific requirements while still ensuring interoperability with other OCPP implementations, provided that the core components have been implemented.

Some implementers choose to incorporate all features specified in OCPP 2.0.1. For example, providers of OCPP software stacks often deliver a comprehensive range of features to their clients. Similarly, Charging Station Management System providers may choose to support a broad spectrum of charging station types across various regions. Conversely, many implementers may only necessitate a subset of the available features, depending on the type of charger being developed (e.g., for light electric vehicles) or the specific use cases to be supported (e.g., remote start functionality).

The OCPP specification includes mechanisms to accommodate the diverse needs of implementers:

1. **OCPP Profiles:** Part 5 of the OCPP specification delineates "Profiles," which define a set of supported functions. The "Core" profile encompasses fundamental OCPP functionalities such as security, authentication, charging session management, and charging station management. Additionally, there are seven other profiles that address advanced features, including advanced security, local list management, smart charging, enhanced user interfaces (e.g., displays), advanced device management, charging station reservation, and ISO 15118 support. The forthcoming OCPP 2.1 version will introduce two additional profiles: V2X (bidirectional power transfer) and DER control (utility management of distributed energy resources).
2. **OCPP Features within a Profile:** This includes both hardware-related and optional features:
    - **Hardware Features:** Specific hardware components of a charging station (e.g., fixed charging cables, credit card readers, or displays) may affect OCPP behavior and determine the applicability of certain OCPP functionalities.
    - **Optional Features:** Numerous behaviors detailed in the OCPP specification are optional. Implementers possess the flexibility to choose which features to support, such as various authentication methods (e.g., RFID, QR code, credit card reader), diverse transaction initiation points (e.g., upon connection of the charging cable or commencement of energy flow), or support for offline transactions.
3. **Product Types for Certification:** The required test suite for certification varies according to the OCPP Product Type:
    - Charging station management systems
    - Charging stations
    - Software stacks for a charging station
    - Mode 1/2-only charging stations (lacking communication with the EV)

These mechanisms provide the EV charging industry freedom to build their charging networks according to their needs, whilst still ensuring interoperability.

This paper investigates, considering resource-constrained devices, the characteristics of a minimum implementation of OCPP 2.0.1 and the resources necessary for its realization.

3. Resource-constrained devices

Public and residential AC charging stations often incorporate a microcontroller for the OCPP communication, for example the Espressif32 or STM32. These microcontrollers are designed for special-purpose systems such as in-field nodes in Internet-of-Things (IoT) networks, but they can also be used as the network controller in charging stations. This class of microcontrollers is constrained in terms of RAM and ROM capacity: usually limited to a few hundred kilobytes of RAM and a few megabytes of ROM.

The OCPP software that is running on a charging station has an impact on the following resources:

- CPU cycles: a series of steps that the CPU goes through to fetch, decode and execute instructions.
- RAM (random access memory): A running application will require working memory for a stack and heap to store data.
- ROM (read-only memory): The OCPP firmware itself is read-only memory.
- Flash memory: This is persistent memory, which is used, for example, to store configuration settings.

Data communication usage does not relate to resource limitations on the device, but we will address it in section Data communication and OCPP messages sizes.

As it turns out, CPU cycles are not a bottleneck as far as OCPP is concerned. OCPP is about transferring information between charging station and backend (CSMS). CPU cycles are mostly used for work related to TLS encryption/decryption and message parsing and not much else.

The most serious constraint will be memory. The firmware is stored in read-only memory (ROM); a smaller firmware image requires less ROM. A running application requires working memory for a stack and heap to store data. This is stored in random-access memory (RAM). The last type of memory is persistent memory, which is used, for example, to store configuration settings. Persistent memory in charging stations often is flash memory.

In this paper we will focus on:

- Minimizing ROM usage (directly related to size of firmware image)
- Minimizing RAM usage (working memory)
- Minimizing persistent memory usage
- Minimizing device model size (affecting both RAM and persistent memory)

In APPENDIX B: Case study MicroOCPP we provide a case study with real-world data of an example minimal implementation of the OCPP protocol stack optimized for the usage on microcontrollers. This data provides insights into how much memory is used for OCPP. It can be used as a basis for estimating the resource requirements of the OCPP interface and selecting microcontrollers according to the memory needs.

APPENDIX C: Case study OpenOCPP describes the ROM and RAM resource usage of a minimal implementation for a complete charging station firmware.

4. Minimizing ROM usage

There are two important factors that help to minimize ROM usage. One is to avoid using an embedded database for the device model. This option is discussed in Minimizing device model size. The other is to minimize the size of the firmware image by reducing the program logic and the associated number of messages that we implement.

The OCPP functionality has been grouped in a number of profiles that represent a set of related use cases. Any charging station needs to support at least the Core profile in order to be eligible for certification. All other profiles are optional. The certification profiles are defined in [OCPP-Part5] and summarized below in table Certification profiles.

Table 1. Certification profiles

Certification Profile	Description
Core	Basic functionality of OCPP
Advanced Security	Advanced security with client certificates
Local Authorization List Management	Support for local white lists
Smart Charging	Support for smart charging with charging schedules
Advanced Device Management	Variable monitoring and custom reports
Advanced User Interface	Support for messages, tariff and cost on display
Reservation	Reservation of an EVSE
ISO 15118 support	Support for ISO 15118-2

In this section we describe the minimal set of use cases and messages that need to be supported for sections A to P of the specification in [OCPP-Part2]. The data is summarized in the table Minimum message set in APPENDIX A: Summary of messages per section.

Section A. Security

Security profile 3 is not part of the Core profile. The messages SignCertificate and CertificateSigned are only needed in the context of security profile 3 and can therefore be omitted.

Required messages:

- SetVariables
- SecurityEventNotification

Section B. Provisioning

In the OCPP 2.0.1 Core profile the only use case in B that is not mandatory is B08 Get Custom Report. This means that the GetReport message does not require implementing. There is no loss in functionality, since all device model information can be retrieved via the GetBaseReport message.

In addition, the messages GetVariables and SetVariables can have a slightly simpler implementation if the number of items per message is set to 1 via ItemsPerMessage[Set/GetVariables].

Required messages:

- BootNotification
- SetVariables
- GetVariables
- GetBaseReport
- NotifyReport
- SetNetworkProfile
- Reset

Section C. Authorization

In the OCPP 2.0.1 Core profile, several authorization methods are described, of which at least one must be implemented. A minimal implementation therefore only has to support a single authorization method. This is likely to be one of C01 (RFID) or C02 (start button), or it only supports remote authorization by CSMS, as described in C04. The use of a local authorization cache to reduce authorization time is optional in the Core profile and can be omitted, since this would require additional persistent storage.

Required messages, one or both of:

- Authorize (for C01 or C02), or
- RequestStartTransaction (for C04)

Section D. Local Authorization List

Section D of the OCPP 2.0.1 specification describes functionality to authorize a user via a local white list. This can be used when the charging station is offline, or to reduce authorization response time when the charging station is online. Storing a local authorization list requires additional persistent storage, and can be omitted for a 'Lite' implementation. Since Local List Management is a separate optional certification profile in OCPP 2.0.1 anyway, a 'Lite' implementation does not differ from a regular OCPP 2.0.1 Core implementation.

Section E. Transactions

To accommodate for the various business models arising in the EV charging industry, OCPP 2.0.1 added flexible start and stop points of a transaction. This does introduce complexity in the firmware though, because the behavior of certain use cases changes depending on when the transaction starts or ends. This complexity can be removed by fixing the start and stop points of the transaction. If both TxStartPoint and TxStopPoint are set to PowerPathClosed then a transaction is started when the user is authorized and the cable is connected, and the transaction stops when authorization ends or cable is disconnected.

This is the same behavior as in OCPP 1.6. The use cases E02 and E03 become equivalent. Use case E04 is not supported, because there is no authorization cache or local authorization list. (See Section C. Authorization and Section D. Local Authorization List.)

Use case E14 (GetTransactionStatus) is required for the "Core" profile of OCPP. It enables the CSMS to request the status of a transaction and to find out whether there are queued transaction-related messages — a situation that may occur when the charger has been offline for a while.

Required messages:

- TransactionEvent
- GetTransactionStatus

Section F. Remote control

Remote control is about remotely authorizing for a transaction and remotely stopping it, remotely unlocking a connector, and triggering a charging station to perform a certain action.

Remote start/stop and unlock connector are required, and may be the only method to start and stop a transaction if local authorization is not implemented. Support for TriggerMessage is optional.

Required messages:

- RequestStartTransaction
- RequestStopTransaction
- UnlockConnector (only in case of a detachable cable)

Section G. Availability

The notification of availability status and heartbeats is part of Core functionality and must be implemented.

Required messages:

- StatusNotification or NotifyEvent
- Heartbeat
- ChangeAvailability

Section H. Reservation

The functionality of Reservation may not be required for resource-constrained devices. Since ‘Reservations’ is a separate optional certification profile in OCPP 2.0.1 anyway, a ‘Lite’ implementation does not differ from a regular OCPP 2.0.1 implementation

Section I. Tariff and Cost

This Functional Block provides tariff and cost information to an EV Driver, if a Charging Station is capable of showing this on a display. It is assumed that resource-constrained devices either do not have a display or have a display that is not actively managed by the CSMS. Therefore, the Tariff and Cost functionality can be omitted. Since this is a separate optional certification profile in OCPP 2.0.1 anyway, a ‘Lite’ implementation does not differ from a regular OCPP 2.0.1 implementation.

Section J. Meter Values

Transaction-related meter values are sent via the TransactionEvent message. There is also a separate MeterValues message that can be used to send clock-aligned meter values outside of a transaction.

Required messages:

- TransactionEvent
- MeterValues

Section K. Smart Charging

Smart charging is not part of the OCPP "core" profile, but there might be a reason to include this in certain minimal implementations anyhow. Home chargers are likely candidates for minimal implementations, and certain in some countries they are required to support smart charging functionality.

A full implementation of smart charging with multiple charging profile kinds and stack levels may be too much for a minimal implementation, so certain limitations are needed in that case:

- The number of charging profiles that a charging station can hold is limited to 1 or 2. That suffices for a TxProfile and/or a TxDefaultProfile.
- The number of periods in a charging schedules is limited to, for example, 3 periods. (Most charging schedules do not have more than 3 periods, anyhow).
- Only a single stack level is supported.

The above reduces memory requirements and simplifies the implementation of a GetCompositeSchedule message.

Required messages (when supporting smart charging):

- SetChargingProfile
- ClearChargingProfile
- GetCompositeSchedule
- Get/ReportChargingProfiles

Section L. Firmware Management

A minimal implementation still needs to be able to update its firmware and do this in a secure manner. Use case L01 Secure Firmware Update is part of OCPP Core profile, and is therefore required in a minimal implementation.

Required messages:

- UpdateFirmware
- FirmwareStatusNotification

Section M. ISO 15118 Certificate Management

Support of ISO 15118 is not expected of a minimal implementation. Some functionality of certificate management, however, of this section is required to implement security profile 2, namely use case M05 InstallCertificate to install the CSMS root certificate.

Required messages:

- InstallCertificate
- GetInstalledCertificateIds
- DeleteCertificate

Section N. Diagnostics

The advanced diagnostic features are part of the optional certification profile ‘Advanced Device Management’. The functionality to download log files (N01 Retrieve Log Information) and to report customer information (N09/N10 Get/Clear Customer Information) are part of the OCPP 2.0.1 Core profile. The use cases N09 and N10 with messages CustomerInformation and NotifyCustomerInformation, however, have a limited use and are not required for normal operation. Functionality to report and clear customer information (N09/N10 Get/Clear Customer Information) was added to adhere to privacy regulation (e.g. GDPR in Europe). If a minimal implementation does not store any customer information, because it does not have the space for this, then implementation of these messages becomes very trivial.

Only use case N01 Retrieve Log Information is required.
Required messages:

- GetLog
- LogStatusNotification
- CustomerInformation
- NotifyCustomerInformation

Section O. Display Messages

This functional block enables an operator to remotely display a message or a cycle of messages on a charging station. It is assumed that resource-constrained devices either do not have a display or have a display that is not actively managed by the CSMS. Therefore, the Display Message functionality can be omitted. Since this is a separate optional certification profile in OCPP 2.0.1 anyway, a ‘Lite’ implementation does not differ from a regular OCPP 2.0.1 implementation

Section P. DataTransfer

This functional block describes functionality to send custom message via the DataTransfer message. DataTransfer messages need not be supported, but the minimal implementation must be able to reject them when received.

5. Minimizing RAM usage

This section provides three ways to reduce the required RAM usage: by limiting the sizes of certain message, by avoiding data duplication, and by reducing the size of the TLS buffer.

5.1. Limiting message sizes

RAM memory is used for stack and heap memory. It is used to store data that is processed, for example while decoding incoming messages or encoding messages to be sent. The amount of memory required to decode messages upon receipt or construct messages before sending can be limited by the following configuration variables:

- DeviceDataCtrlr.ItemsPerMessage
- DeviceDataCtrlr.BytesPerMessage
- DeviceDataCtrlr.ReportingValueSize
- MonitoringCtrlr.ItemsPerMessage
- MonitoringCtrlr.BytesPerMessage
- SmartChargingCtrlr.PeriodsPerSchedule
- SampledDataCtrlr.TxStarted/Updated/EndedMeasurands
- AlignedDataCtrlr.Measurands

These configuration variables limit the size of potentially very large messages.

5.2. Avoiding data duplication in RAM

Data transmission in an OCPP implementation takes several steps. For incoming messages, the data is first received on the networking interface, then the TLS library decrypts it, providing the data to a WebSocket client which buffers all data until a JSON message is complete. Depending on the architecture of the OCPP implementation, the JSON message could further be copied and forwarded to message listeners until it is finally consumed. It looks similar in the other direction. This multistep pipeline potentially means that a buffer between any two steps holds a full copy of the payload data sent via OCPP, which is a common pitfall of implementing OCPP.

There is a significant optimization potential in avoiding copying the unencrypted data in RAM. For incoming messages, that would mean keeping the JSON message in the TLS input buffer, instructing the JSON library to parse the JSON object in place and only copying data when updating the final data model. Of course, the exact approach of avoiding data duplication depends on the software architecture but should be similar to this description. For outgoing messages, the same idea should be considered.

It is important to pay special attention to the heap consumption behavior during receipt and sending of messages. Usually, the heap occupation spikes during message processing and for microcontrollers, the worst-case heap occupation is the most relevant aspect to optimize for.

5.3. Reducing TLS communication buffer

A TLS communication by default requires two 16 kB buffers. This buffer can be reduced by adapting the TLS fragment length.

TLS involves sending "Records" between peers. Records can be of type “Handshake”, “Alert”, “ChangeCipherSpec”, “Heartbeat” or “Application”. OCPP messages are sent in Application records. The payload contains a “fragment” of the application data. The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^14 bytes (16kB) or less.

TLS peers need to maintain an input and an output buffer to store an entire fragment of 16 kB. For a low resource device it is a large cost to allocate 32 kB for the TLS connection.

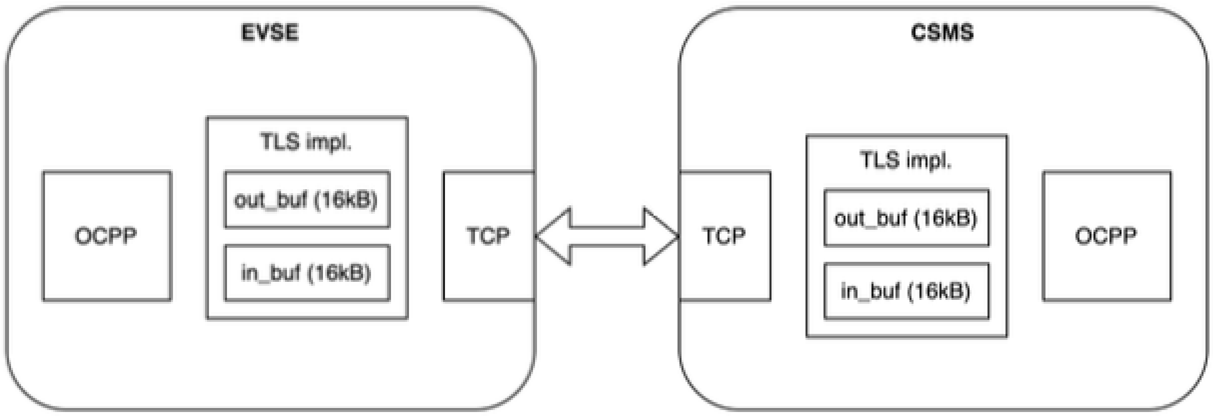

Figure 1. Peers allocating standard 16 kB TLS buffers

A TLS extension is defined in TLS Extensions RFC6066 Section 4, that allows the client to ask for a different maximum fragment length than the default 16kB. A client can ask for a maximum fragment length of 0.5 kB, 1 kB, 2 kB or 4 kB. This TLS extension is, however, not widely supported and native managed cloud TLS termination services typically don’t support this.

A resource-constrained Charging Station SHOULD try to negotiate a smaller TLS maximum fragment size, and if that is not accepted by the peer, then Charging Station MAY unilaterally decide to allocate less memory to its TLS output buffer. A TLS maximum fragment length of 2 kB is suggested based on data collection during certification tests, which shows that 99% of the messages fit in a 2 kB buffer. This is described in the next section.

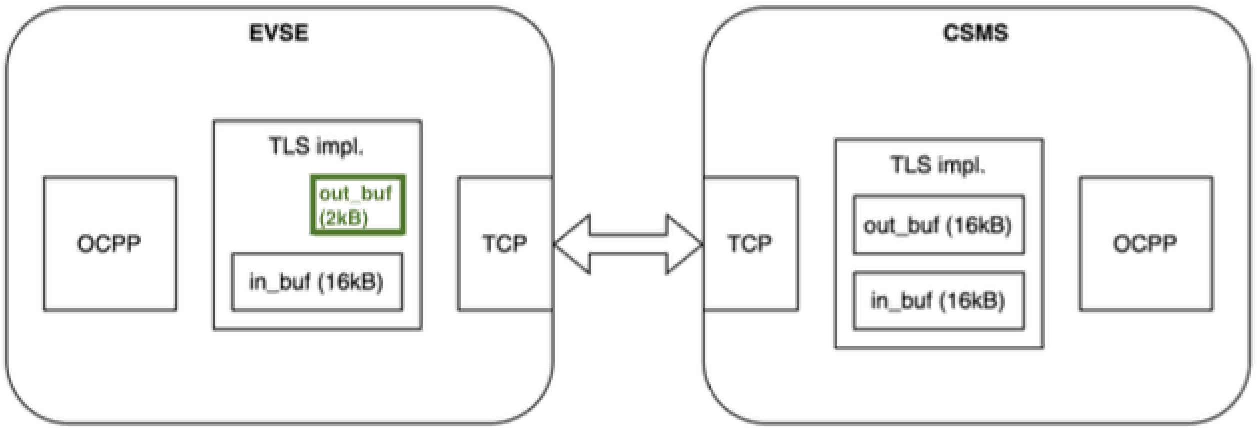

Figure 2. Charging Station allocating a 2 kB TLS output buffer

5.4. Data communication and OCPP messages sizes

The data communication usage of a charging station is highly dependent on its usage profile and configuration. A residential charging station with one charging session per day generates significantly less data traffic than a busy fast charger. Similarly, some chargers may be configured to provide meter values every 15 minutes, while a fast charger might be set to do this every 30 seconds. Therefore, it is not possible to provide a general estimate of data transmission without specifying the usage profile of the charging station.

There are no standardized usage profiles for charging stations that allow for an estimated data usage. To offer some insight into the size of OCPP messages, data logs from official OCPP certification tests of 15 charging stations were analyzed. A total of 931 test cases were executed, resulting in the transmission of 32,636 messages.

Example of an OCPP message

```
 [2,"38a28818-3a91-4e82-a03a-fbd992a92b11", "TransactionEvent",
{"eventType":"Started","evse":{"connectorId":1,"id":1},"idToken":{"idToken":"
00abf91f","type":"ISO14443"},"seqNo":0, "timestamp":"2024-04-
23T02:06:02.444Z","transactionInfo":{"chargingState":"Idle","transactionId":"
a32ad195-e467-4423-ae0b-61e1e617f1b1"},"triggerReason":"Authorized"}]
```

From this data, the following statistics were derived:

- Smallest message length: 25 characters
- Average message length: 235 characters
- Largest message length: 64,907 characters

NOTE The largest message resulted from one charging station sending an entire device model report as a single NotifyReport message, rather than breaking it into smaller chunks as is typically done.

The majority of OCPP messages (~95%) are smaller than 500 characters. The distribution of larger OCPP messages is as follows:

Message size	Message count	Percentage of total (32636)
> 500 chars	1334	4%
> 1000 chars	306	0.94%
> 2000 chars	148	0.45%
> 4000 chars	34	0.10%

6. Minimizing persistent memory usage

6.1. Persistence of charging profiles

OCPP 2.0.1 requires that charging profiles are stored persistently. If a CSMS sends frequent updates of a charging profile, this will eventually wear out flash memory. This is especially concerning for a charging profile of type TxProfile, because that is sent for every transaction. A similar concern applies to charging profile of type ChargingStationExternalConstraints, which originate from an external actor, such like an energy management system.

For this reason, the requirement for persistence of TxProfile and ChargingStationExternalConstraints charging profiles has been removed from OCPP 2.1, reducing the amount of persistent memory needed.

6.2. Minimizing device model size

The OCPP device model is a repository of configuration variables (stored in controller components) and components that represent their physical counterparts in the charging station.

A substantial amount of ROM and persistent memory can be saved if the device model is implemented without utilizing an embedded database, and if only mandatory components are supported. For the limited functionality of resource-constrained charging stations, this is a viable option.

Of the more than 50 required variables for OCPP 2.0.1, only 19 require to be writable by CSMS and need to be stored in persistent memory. These are configuration variables that will not be changed often and can therefore easily be stored in a simple file without requiring an embedded database. The remaining variables are either read-only (preconfigured) values that can be part of the firmware, or are real-time state values, like "Availability", that do not require to be stored.

6.2.1. Mandatory charging infrastructure components

Charging infrastructure components refer to physical components that are used for charging:

- ChargingStation
- EVSE
- Connector

The variables that are required by the OCPP specification for these components can all be treated as read-only variables, which means that their value can be hard-coded in the implementation, either as a fixed value or as a value that is reported based on the system state.

The SupplyPhases of an EVSE can be hard-coded in a configuration file, because it will not change during operation. The AvailabilityState (Available, Occupied, etc.), will change during operation, but the current state can readily be reported by the application at any time. There is no need to store that in a database.

ChargingStation

- AvailabilityState: reports current state from firmware (read-only)

- Available: set to true (read-only)
- SupplyPhases: preconfigured value reported from firmware (read-only)

EVSE

- AvailabilityState: reports current state from firmware (read-only)
- Available: set to true (read-only)
- SupplyPhases: preconfigured value reported from firmware (read-only)
- Power: only maxLimit is required, but actual value can also be reported from firmware (read-only)

Connector

- AvailabilityState: reports current state from firmware (read-only)
- Available: set to true (read-only)
- SupplyPhases: preconfigured value reported from firmware (read-only)
- ConnectorType: set to fixed value as part of firmware (read-only)

6.2.2. Controller components

This section lists the device model components that are required to be implemented even in a minimal implementation. The majority of these variable can be read-only and can therefore be hard-coded in the firmware. The variables that must be configurable by CSMS are shown in **boldface**.

AlignedDataCtrlr

The AlignedDataCtrlr is required for the Core profile, but in most cases only the use of SampledDataCtrlr will be enough. Its variables are required to be Read/Write, but can initially be set to values that will cause no MeterValues to be reported for aligned data measurands. This means that:

- Available: set to true
- **Interval**: set to 0 (no aligned data to be reported)
- **Measurands**: set to empty
- **TxEndedMeasurands**: set to empty (no aligned data at end of transaction)
- **TxEndedInterval**: set to 86400 (24 hours) (Exact value irrelevant if TxEndedMeasurands is empty)

AuthCtrlr

The AuthCtrlr has three required parameters:

- AuthorizeRemoteStart: can be read-only and set to a fixed value, depending on whether remote start is supported or not.
- LocalAuthorizeOffline, LocalPreAuthorize: can be read-only. Value is irrelevant, because local authorization cache or authorization list are not supported in a minimal implementation.

ClockCtrlr

The ClockCtrlr is only required to present the current time and time source.

- DateTime: a read-only value that reflects current clock time.
- TimeSource: read-only and set to Hearbeat, because a minimal implementation only supports heartbeat as time source.

DeviceDataCtrlr

Purpose of the DeviceDataCtrlr is to report limits on messages from a CSMS for device model variables, which can potentially be very large.

- ItemsPerMessage[GetReport/GetVariables/SetVariables]: read-only, set to a fixed small enough value, e.g. 1.
- BytesPerMessage[GetReport/GetVariables/SetVariables]: read-only, set to a fixed small enough value, e.g. 300.

OCPPCommCtrlr

The OCPPCommCtrlr has a set of required variables, some of which must be writable by CSMS. This can be stored in a file that holds the configuration settings.

- FileTransferProtocols: is read-only and set to the supported value(s).
- MessageTimeout[Default]: is read-only and set to configured value.
- **MessageAttemptInterval**[TransactionEvent]: is read-write and supported.
- **MessageAttempts**[TransactionEvent]: is read-write and supported.
- **NetworkConfigurationPriority**: is read-write and supported.
- **NetworkProfileConnectionAttempts**: is read-write and supported.
- **OfflineThreshold**: is read-write and supported.
- **ResetRetries**: is read-write and supported.
- UnlockOnEVSideDisconnect: can be read-only and set to supported value.

SampledDataCtrlr

The SampledDataCtrlr defines the measurands to report during a transaction. The values must be writable by CSMS and therefore need to be stored in a configuration file.

- **TxEndedInterval**: : is read-write and supported.
- **TxEndedMeasurands**: is read-write and supported.
- **TxUpdatedInterval**: is read-write and supported.
- **TxUpdatedMeasurands**: is read-write and supported.

SecurityCtrlr

The SecurityCtrlr holds properties related to the websocket connection and certificaties.

- CertificateEntries: can be read-only set to number of supported certificates.
- **OrganizationName**: read-write and set by CSMS.
- SecurityProfile: can be read-only and reflects current security profile.
- **Identity**: is not required, yet recommended to be supported as a read-write variable, because it is part of the connection URL to CSMS.

SmartChargingCtrlr

Smart charging is not part of a basic implementation, but it may be desired to include in a minimal implementation in certain cases. The following is a minimal SmartChargingCtrlr:

- Available: set to true or false (read-only)
- Entries[ChargingProfiles]: set to number of supported charging profiles, e.g. 1 (read-only)
- **LimitChangeSignificance**: needs to be writable by CSMS
- PeriodsPerSchedule: set to number of periods supported, e.g. 3 (read-only)
- ProfileStackLevel: set to highest stack level supported, e.g. 0 for only 1 level (read-only)
- RateUnit: set to rate unit supported, e.g. A (read-only)

TxCtrlr

The TxCtrlr defines start and stop point of a transaction. For a minimal implementation this is best set to PowerPathClosed, which is the same condition that is used in OCPP 1.6.

- **EVConnectionTimeout**: must be writable by CSMS
- StopTxOnEVSideDisconnect: read-only, set to true
- **StopTxOnInvalidId**: must be writable by CSMS
- TxStartPoint: can be set to read-only and recommended value PowerPathClosed.
- TxStopPoint: can be set to read-only and recommended value PowerPathClosed.

7. Implementation notes

7.1. Transaction model

OCPP 2.0.1 expands upon the 1.6 transaction model by introducing (among other things) flexible start and stop points and significantly expanding specific requirements for individual use cases. While this improvement increases the specification’s flexibility and precision, it also complicates the process of translating the specification into concrete implementations and defining reasonable abstractions and requirements for operation-level request handlers (for example: collecting all use case requirements into a single "onRequestStartTransactionRequest" handler within an implementation). To address this challenge, embedded manufacturers are encouraged to reference existing open-source implementations and associated designs as potential models for reuse.

In this section we present one possible approach to implementing OCPP 2.0.1 transaction handling with minimal memory usage. The implementation notes will focus on an approach only targeting the minimum recommended support from Section E. Transactions; that is TxStartPoint and TxStopPoint both set to PowerPathClosed with no support for other transaction start/stop points.

7.1.1. Authorization

An authorization instance represents one of the following:

- A RequestStartTransactionRequest sent from the CSMS
- An RFID tap on the station not already associated with a transaction
- A custom vendor-specific "autostart" extension

When any of these operations occur, a authorization instance is created, which will:

- Send out an AuthorizeRequest if necessary to authorize the transaction with the CSMS
- Be matched against a plugged-in connector to start a transaction
- Expire after a timeout (such as ConnectionTimeOut)

The authorization instance is removed when:

- It is used to start a transaction, or
- It is replaced by a more recent similar request

Note: it may be reasonable for an implementation to only allow a authorize operation on the station as a whole (so that an RFID tap or a RequestStartTransactionRequest would replace any earlier attempts), or to allow multiple requests to co-exist in a limited fashion. For example, it may be reasonable for two RequestStartTransactionRequests for different EVSE IDs (on a multi-connector charger) to co-exist with each other, but it would likely be confusing to allow two RFID taps or an RFID tap and a RequestStartTransactionRequest with no EVSE ID to co-exist with each other.

7.1.2. Transactions

A transaction represents an on-going (persistent) operation that has been resolved to a single connector. In this model transactions represent authorised charging sessions and while a transaction instance is active energy is offered to the vehicle (according to the active power management rules).

While transactions are running various transactions events are sent out in response to the following events:

- When a transaction starts/stops
- When the charging status changes (for example from Idle to Charging or vice-versa)
- When a transaction is de-authorised by the CSMS
- When meter value readings are added (based on the OCPP configuration)

All of these events must be persisted and replayed to the CSMS in the event the charger goes offline or a power loss occurs. An effective way to do this without dramatically impacting transaction handling is to push transaction events into a **pending message queue** which queues those messages and delivers them to the CSMS, retrying and persisting as necessary.

7.1.3. Pending message queue

Pending messages can be accumulated in a separate module for storage/retries (see, for example: **E11**). This is useful to separate the complexity of the transaction state machine from:

- Sending messages
- Retrying historic messages
- Discarding old data due to memory or storage space limitations
- Persisting historic messages to prevent loss during power outages

One approach to this type of separation is the following:

- Augment pending (transaction) messages with a transaction ID and a "removal priority"
- Remove messages to save space as per E04.FR.08, E11.FR.05, and E12.FR.05:
    - First remove meter value data in the middle of a transaction (preserving the first and last meter value as long as possible)
    - Then remove the first and last meter value

Note that this approach typically comes with per message overheads on the order of hundreds of bytes per message. This can rapidly eat into available heap/flash space when attempting to store these records. One approach that was found to be effective in practice was to compress the data using standard gzip compression in memory into reasonable sized blocks. This dramatically improved storage (record size was reduced from roughly 600 bytes uncompressed to roughly 45 bytes compressed) at the cost of some processing time and fixed heap costs while compressing/decompressing (for gzip internals) when attempting to process/remove records (required recompression of the block). The results of one such implementation was the following:

- ZLib window bits: 11

- ZLib memory level: 1
- Compression overhead: ~15k heap
- Decompression overhead: ~7k heap
- Average decompressed message size: 640.1 bytes
- Average compressed message size: 51.6 bytes

Other approaches might be:

- Binary encoding of messages internally
- Custom delta encoding of data in stream

These approaches may offer a lighter message-specific implementation that may be less resource intensive or may achieve improved compression ratios.

7.2. Supporting HTTPS in FirmwareUpdate, GetLog, or third-party connections

The 2.0.1 specification allows a manufacturer to support a variety of secure and insecure protocols that it accepts as a communication channel for FirmwareUpdate/GetLog connections, such as HTTP, HTTPS, FTP, FTPS, and SFTP. It is clearly beneficial to use common secure protocols here, particularly because both firmware updates and log messages may contain sensitive data — however that choice often comes with significant additional costs in terms of memory usage.

Including a FTPS or SFTP library may involve attempting to pull in large popular open source libraries like libcurl that can be difficult and expensive to integrate into an embedded environment, while HTTPS support often raises questions about certificate management overhead, complexity, and so on, especially in resource constrained lite devices. Note that endpoints provided in those requests (specifically those based on TLS like HTTPS or FTPS) may not use TLS certificates signed by root CAs provided to the charger to secure its CSMS connection in security profile 2/3, and the specification has not yet defined how a manufacturer should secure these TLS connections.

One alternative that was found to be effective is to support HTTPS connections to these "external" URLs and secure the connections using the certificate bundle support provided by ESP-IDF/Mbed-TLS. This was found to effectively allow secured HTTPS connections with the same level of complexity as supporting HTTP connections, no significant heap overhead, and only a small impact on binary size (approximately 62 KiB, which was small next to the reference firmware sizes of roughly 1 MiB - 1.5 MiB). Similar approaches at varying levels of complexity can likely be adapted to other embedded platforms supporting the Mbed-TLS library (depending on the platform's level of support for this feature). It's also worth noting that this approach may be applicable for securing other third-party HTTPS connections a charging station may need to establish outside of the OCPP protocol.

The following settings were used to enable certificate bundle support on a reference ESP32 implementation:

- CONFIG_MBEDTLS_CERTIFICATE_BUNDLE=y
    - ▯ Enables the CA bundle

- CONFIG_MBEDTLS_CERTIFICATE_BUNDLE_DEFAULT_FULL=y
    - Embeds a complete list of Mozilla’s NSS root certificates
    - Impact on binary size: ~64057 bytes
- config.crt_bundle_attach = esp_crt_bundle_attach;
    - Uses the CA bundle to verify the server’s certificate in the standard ESP-IDF HTTPS client

IMPORTANT	CA bundles should not be used to secure the charging station’s OCPP connection to a back-end in security profile 2/3. Trusting a large number of CA authorities in security profile 2/3 is not considered secure.

References

- [OCPP-Part2] OCPP 2.0.1 Part 2 - Specification
- [OCPP-Part5] OCPP 2.0.1 Part 5 - Certification Profiles
- [OCPP-Part6] OCPP 2.0.1 Part 6 - Test Cases
- [OCPP-16] OCPP 1.6
- [Sec-WP] OCPP 1.6 Security Whitepaper (3rd edition)
- [MicroOCPP] OCPP 1.6/2.0.1 protocol stack, https://github.com/matth-x/MicroOcpp
- [OpenOCPP] OCPP 1.6/2.0.1 implementation by ChargeLab, planned to become open source

APPENDIX A: Summary of messages per section

Table 2. Minimum message set

Section	Use Cases	Messages
A. Security	• A00 Security Profile 1 & 2 • A01 Update CS password • A04 Security Event	SetVariables, SecurityEvent
B. Provisioning	• B01/B02/B03 Cold Boot - Accepted, Pending, Rejected • B04 Offline Idle • B05 Set Variables • B06 Get Variables • B07 Get Base Report • B09 Set Network Profile • B10 Migrate to New CSMS • B11/B12 Reset Charging Station	BootNotification, SetVariables, GetVariables, GetBaseReport, NotifyReport, SetNetworkProfile, Reset
C. Authorization	• C01, C02, or C04 Authorization	Authorize
(D. Local Authorization List)	-	
E. Transactions	• E01 Start Transaction (PowerPathClosed) • E05 Start Transaction - id not accepted • E06 Stop Transaction (PowerPathClosed) • E07 Transaction stop by idToken • E08 Transaction stop while offline • E09 Transaction Stop - cable disconnect on EV • E11 Connection Loss during transaction • E12 Inform of offline transactions • E13 Transaction message not accepted	TransactionEvent, GetTransactionStatus

Section	Use Cases	Messages
F. Remote Control[1]	• (F01 Remote Start - Cable First) • (F02 Remote Start - Start First) • (F03 Remote Stop) • (F05 Remote Unlock) • (C05 Authorization CSMS Initiated Transactions)	RequestStartTransaction, RequestStopTransaction, UnlockConnector
G. Availability	• G01 Status Notification • G02 Heart Beat • G04 Change Availability Charging Station	StatusNotification, Heartbeat, ChangeAvailability
(H. Reservation)	-	
(I. Tariff and Cost)	-	
J. Meter Values	• J02 Transaction-Related Meter Values	TransactionEvent, MeterValues
(K. Smart Charging)*	• (K01 Set Charging Profile) • (K02 Central Smart Charging) • (K10 Clear Charging Profiles)	(SetChargingProfile), (ClearChargingProfiles), (GetCompositeSchedule), (GetChargingProfiles), (ReportChargingProfiles)
L. Firmware Management	• L01 Secure Firmware Update, • (L02 Non-Secure Firmware Update)	UpdateFirmware, FirmwareStatusNotification
M. Certificate Management	• M03 Retrieve list of available certificates • M04 Delete a certificate • M05 Install CA certificate	InstallCertificate, GetInstalledCertificateIds, DeleteCertificate
N. Diagnostics	• N01 Retrieve Log Information	GetLog, GetCustomerInformation, ClearCustomerInformation
(O. Display messages)	-	
(P. Datatransfer)	-	

* *Not part of Core profile, but possibly desired functionality in a minimal implementation*

APPENDIX B: Case study MicroOCPP

NOTE The metrics of MicroOCPP have been provided by Matthias Akstaller and are based on the state of development of MicroOCPP as of July 2025.

MicroOCPP (see [MicroOCPP]) is an open source implementation of an OCPP protocol stack in C/C++ for OCPP 1.6 and OCPP 2.0.1. MicroOCPP can be compiled as OCPP 1.6, OCPP 2.0.1 or both. It provides an API to interface with the firmware that controls the charging station. The metrics in this appendix are only about the OCPP protocol stack.

This appendix presents a size analysis of MicroOCPP which shows the impact of various OCPP 2.Lite features on the total size of the firmware. The numbers provide a reference point for own estimations on the final ROM space required by an OCPP 2.Lite implementation.

ROM requirements for a minimal message set

The reference implementation for OCPP 2.0.1 has a binary size of ~120 kB in total. For the benchmarks, it was compiled with -Os, no RTTI or exceptions and newlib as the standard C library. A size profiler tool (bloaty) was used to calculate how much binary size each compilation unit adds to the firmware. In the example OCPP 2.Lite implementation shown here, each message type is a separate compilation unit.

The following functionality, that exists in OCPP 1.6, has not yet been ported to the OCPP 2.0.1 implementation:

- LocalAuthListManagement
- Reservation
- FirmwareManagement
- DataTransfer

Despite those limitations, the numbers are overall significant. If we assume the same size as in OCPP 1.6 for the functionality that is missing in OCPP 2.0.1, then the full scope of OCPP 2.Lite is only ~10% larger than OCPP 1.6. We are confident that the full scope of OCPP 2.Lite remains well under 200 kB of firmware size.

Table 3. Module sizes in bytes for MicroOCPP as OCPP 1.6, OCPP 2.0.1 or both

Module	OCPP 1.6	OCPP 2.0.1	Both enabled
General library functions	35388.0	34852.0	38804.0
A - Security	700.0	732.0	804.0
B - Provisioning - Configuration (v16)	8928.0		9008.0
B - Provisioning - Variables (v201)		18060.0	18004.0
B - Provisioning - Other	3536.0	4300.0	5848.0
C - Authorization	932.0	868.0	1640.0
D - LocalAuthorizationListManagement	5532.0	t.b.d.	5568.0
E - Transactions	17016.0	24608.0	40948.0
F - RemoteControl	3328.0	3440.0	4916.0

Module	OCPP 1.6	OCPP 2.0.1	Both enabled
G - Availability	5996.0	3852.0	8776.0
H - Reservation	3672.0	t.b.d.	3728.0
J - MeterValues	10432.0	5668.0	13452.0
K - SmartCharging	11768.0	12136.0	13400.0
L - FirmwareManagement	4944.0	t.b.d.	4984.0
M - CertificateManagement	6420.0	6364.0	6412.0
N - Diagnostics	7793.0	5745.0	8057.0
P - DataTransfer	472.0	t.b.d.	476.0
Total	126857.0	120625.0	184825.0

In OCPP 2.0.1 the sending of transaction-related meter values is part of section E - Transactions. For a proper comparison between OCPP 1.6 and OCPP 2.0.1 one needs to sum E - Transactions and J - MeterValues in the table above, as shown below:

Table 4. Module sizes in bytes for MicroOCPP for transactions (E + J)

Module	OCPP 1.6	OCPP 2.0.1	Both enabled
E - Transactions	17016.0	24608.0	40948.0
J - MeterValues	10432.0	5668.0	13452.0
Total	27448.0	30276.0	54400.0

An important improvement of OCPP 2.0.1 is the revised transaction mechanism, which addresses the often insufficient flexibility of the version 1.6 Transactions. Most importantly, the backend can now define the start and stop point and query the current transaction status while the charger provides more transaction-related information in intermediate TransactionEvent messages.

In the MicroOCPP reference implementation, the firmware size in Table 2 increases by 10% for OCPP 2.0.1, but the complexity of the implementation can be reduced significantly, if the implementation would use a fixed value, e.g. PowerPathClosed, as both start and stop point, as is recommended in Section E. Transactions.

RAM requirements

The RAM usage of the OCPP interface is determined by the concrete implementation of the protocol. To allow for an estimation of how much memory such an implementation would need, we present the memory usage statistics of an example OCPP 2.Lite implementation.

The following table shows the peak memory usage for OCTT test cases which are mentioned in the first column. These are test cases from [OCPP-Part6] and executed using the OCTT testing tool from Open Charge Alliance. The test cases that have passed OCTT testing are marked with "x" in the table. Test cases marked with "-" have not passed OCTT testing. MicroOCPP will be updated in the coming months to pass all test cases in the table. As a result the recorded heap usage for these test cases may change as the implementation is updated.

For the full and updated test results, see the official benchmarks page in [MicroOCPP].

Table 5. RAM usage for an OCPP 2.0.1 implementation

Testcase	**Pass**	**Heap usage (Bytes)**
B:Provisioning		
Get Variables - single value	x	2052
Get Variables - multiple values	-	2052
Set Variables - single value	x	2052
Set Variables - multiple values	-	2387
Set Variables - invalidly formatted values	-	1540
Get Base Report - ConfigurationInventory	x	5855
Get Base Report - FullInventory	x	6065
Get Variables - Unknown component	x	1459
Get Variables - Not supported attribute type	x	1412
Set Variables - Unknown component	x	1459
Set Variables - Unknown variable	x	1397
Set Variables - Not supported attribute type	x	1536
Set Variables - Read-only	-	1536
C:Authorization		
Local start transaction - Authorization Invalid/Unknown	x	2973
Local Stop Transaction - Different idToken	-	3091
Local start transaction - Authorization Blocked	x	2973
Local start transaction - Authorization Expired	x	2973
Stop Transaction with a Master Pass - Without UI	-	3091
E:Transactions		
Start transaction options - PowerPathClosed	-	3091
Start transaction options - EnergyTransfer	x	3090
Local start transaction - Cable plugin first - Success	-	2835
Local start transaction - Authorization first - Success	-	3091
Local start transaction - Authorization first - Cable plugin timeout	-	3091
Local Stop Transaction - Accepted	-	3091
Stop transaction options - PowerPathClosed - Local stop	-	3091
Start transaction options - EVConnected	-	2021
Start transaction options - Authorized - Remote	-	2946
Stop transaction options - StopAuthorized - Local	-	3086
Stop transaction options - Deauthorized - EV side disconnect	-	3086

Testcase	**Pass**	**Heap usage (Bytes)**
Stop transaction options - EVDisconnected - EV side (able to charge IEC 61851-1 EV)	-	3087
Stop transaction options - StopAuthorized - Remote	-	3086
Disconnect cable on EV-side - Deauthorize transaction - UnlockOnEVSideDisconnect is true	-	3091
Disconnect cable on EV-side - Deauthorize transaction - UnlockOnEVSideDisconnect is false	-	3091
Check Transaction status - TransactionId unknown	-	1397
Check Transaction status - Transaction with id ongoing - with message in queue	-	1397
Check Transaction status - Transaction with id ongoing - without message in queue	-	3091
Check Transaction status - Transaction with id ended - with message in queue	-	1397
Check Transaction status - Transaction with id ended - without message in queue	-	3091
Check Transaction status - Without transactionId - with message in queue	-	1397
Check Transaction status - Without transactionId - without message in queue	-	1397
Stop transaction options - PowerPathClosed - Remote stop	-	3091
Stop transaction options - Deauthorized - timeout	x	2973
F:Remote Control		
Remote start transaction - Cable plugin first	-	2951
Remote start transaction - Remote start first - AuthorizeRemoteStart is true	-	1536
Remote start transaction - Remote start first - AuthorizeRemoteStart is false	-	3091
Remote start transaction - Remote start first - Cable plugin timeout	-	2048
Remote unlock Connector - With ongoing transaction	-	3091
Remote unlock Connector - Without ongoing transaction - Accepted	-	1397
Remote unlock Connector - Without ongoing transaction - No cable connected	-	1397
Remote stop transaction - Success	-	3091
Remote stop transaction - Rejected	-	3091
Remote unlock Connector - Without ongoing transaction - UnknownConnector	x	1397
Trigger message - MeterValues - Specific EVSE	-	1536
Trigger message - MeterValues - All EVSE	-	1536
Trigger message - TransactionEvent - Specific EVSE	-	3121
Trigger message - TransactionEvent - All EVSE	-	1412
Trigger message - Heartbeat	x	1397
Trigger message - StatusNotification - Specific EVSE - Available	x	1397
Trigger message - StatusNotification - Specific EVSE - Occupied	x	1397

Testcase	**Pass**	**Heap usage (Bytes)**
Trigger message - BootNotification - Rejected	x	1397
Trigger message - NotImplemented	x	1397
G:Availability		
Connector status Notification - Available to Occupied	x	1397
Connector status Notification - Occupied to Available	x	1597
Change Availability EVSE - Operative to inoperative	x	1397
Change Availability EVSE - Inoperative to operative	x	1397
Change Availability Charging Station - Operative to inoperative	x	1397
Change Availability Charging Station - Inoperative to operative	x	1397
Change Availability Connector - Operative to inoperative	x	1397
Change Availability Connector - Inoperative to operative	x	1397
Change Availability EVSE - Operative to operative	x	1397
Change Availability EVSE - Inoperative to inoperative	x	1397
Change Availability EVSE - With ongoing transaction	-	3091
Change Availability Charging Station - Operative to operative	x	1397
Change Availability Charging Station - Inoperative to inoperative	x	1397
Change Availability Charging Station - With ongoing transaction	-	3091
Change Availability Connector - Operative to operative	x	1397
Change Availability Connector - Inoperative to inoperative	x	1397
Change Availability Connector - With ongoing transaction	-	3091
J:Meter Values		
Sampled Meter Values - EventType Started - EVSE known	-	2865
Sampled Meter Values - Context Transaction.Begin - EVSE not known	-	3116
Sampled Meter Values - EventType Updated	-	1397
Sampled Meter Values - EventType Ended	-	1397
Simulator stats		
Base memory occupation		15430
Test case maximum		6065
Total memory maximum		21802

The above table does not include memory usage for TLS and WebSockets. This may vary based on the TLS and WebSocket libraries used. For ways to reduce the TLS memory overhead, refer to paragaph 4.3. Reducing TLS communication buffer.

To support all the test cases of the table above with some added headroom, the RAM requirement would be about 50 kB for the OCPP 2.Lite client.

Memory requirements for Variables

MicroOCPP defines a variable class and instantiates one memory object per variable, so that the memory usage increases linearly with the device model size.

As suggested in Minimizing device model size, the example implementation stores persistent variables in a plain-text format on flash memory.

Mutability	**RAM size per Variable**	**Flash size per Variable**
Read-only	94 B	0 B
Read-write	142 B	107 B

In the measurement results, read-only variables were of a constant size, because the actual data was stored in ROM and not copied to RAM. Read-write variables need to keep the value in RAM, which increases the total size.

In this table, we took the worst-case scenario, which are free-text string values, and assumed a length of 40 characters. The flash size is based on an example key-value data structure.

APPENDIX C: Case study OpenOCPP

NOTE The appendix about OpenOCPP has been provided by Cedric Shui of ChargeLab.

OpenOCPP (see [OpenOCPP]) is an open source multi-platform combined OCPP 1.6 and OCPP 2.0.1 implementation produced by ChargeLab targeting the full core 2.0.1 requirements (not restricted to 2.lite requirements in this document). It serves as a useful point of comparison for general embedded implementations.

ROM requirements

Total ROM overhead of a feature complete charger binary supporting OCPP 1.6/2.0.1 was found to be approximately 1.6 MiB which was deemed tolerable for the target platform (an ESP32 microcontroller with 4 MB of flash) and was not optimized further. It's worth noting that approximately 489 KB is consumed by the primary application object while the remainder (~1.1 MB) is consumed by the various ESP-IDF platform libraries necessary to run the firmware. This again highlights the impact of platform versus library overheads.

An attempt was made to categorize the ROM usage of the "OCPP library" portion of the implementation to provide implementers with a sense of the size of the OCPP implementation in isolation (without including platform and non-OCPP costs). This was found to be sensitive to where we drew this boundary around our implementation, however we were able to obtain a reasonable upper and lower bound of ~378 KB and ~182 KB respectively (for OpenOCPP's combined 1.6 and 2.0 implementation).

RAM requirements

The RAM usage of individual components was analyzed by logging platform-level aggregate heap usage across initialisation and, in some cases, augmenting them with data from known idle-state accumulated buffers. Additional, for the sake of comparison the impact of ESP-IDF **system** components was included as well:

Table 6. RAM usage by component

Sections	Idle RAM Usage
Security	Platform module: 304 bytes Note: does not include contributions from initializing ESP-IDF subsystems (wifi, file-system, etc).
Provisioning	Boot notification module: 208 bytes Configuration module: 96 bytes Reset module: 96 bytes **Total**: 400 bytes

Sections	Idle RAM Usage
Authorization **Transactions** **Remote Control** **Meter Values**	Pending messages module: 440 bytes Transaction module (1.6): 344 bytes Transaction module (2.0): 452 bytes **Total**: 1236 bytes Note: heap usage of pending messages module will scale up to 10 KB (of compressed data) to accommodate transaction data while offline.
Availability	Connector status module: 536 bytes Heartbeat module: 160 bytes **Total**: 696 bytes
Smart Charging	Power management module (1.6): 196 bytes Power management module (2.0): 912 bytes **Total**: 1108 bytes Note: heap usage of power management modules will scale based on the number of installed profiles, approximately according to the size of the SetChargingProfileRequest objects.
Firmware Management	Firmware management module: 376 bytes
Diagnostics	Get logs module: 3080 bytes Note: retains up to 2 KB (or more as configured by the manufacturer) of compressed log messages in-memory for the purpose of responding to GetLog requests.

Table 7. RAM usage additional elements

Name	Idle RAM Usage	Notes
Settings	5936 bytes	Defines 126 standard and internal settings for an average per setting heap size of ~47 bytes.
Portal	112 bytes	Does not include HTTP/HTTPS overheads from the ESP-IDF platform (see below)
OcppMessageHandler 1.6	748 bytes	Internal component related to OCPP 1.6 message processing
OcppMessageHandler 2.0	584 bytes	Internal component related to OCPP 2.0 message processing
Wifi/Core	42560 bytes	Include default ESP-IDF event loop creation and wifi STA/AP initialization.
Websocket	25268 bytes	Includes initializing and connecting to a single TLS-based websocket connection using esp_websocket_client

Name	Idle RAM Usage	Notes
HTTPS Server	18780 bytes	Initializing ESP-IDF HTTPS server using self-signed certifiate for provisioning/control
HTTP Server	9988 bytes	Initializing ESP-IDF HTTP server for provisioning/control

It’s worth noting that the majority of heap contribution here comes from **platform internals** with a sufficiently tuned implementation. It is worth noting that many of the implementations/approaches that were deemed reasonable for a 1.6 implementation dramatically increased heap usage within the application and required substantial restructuring.

Device model variable size optimizations

The settings definitions went through several iterations to achieve the final heap requirements of approximately 47 bytes per variable. The primary optimizations that achieved that in our implementation were the following:

- Sharing a single large explicit setting container among modules with concrete variable definitions. This eliminated the overhead in maintaining per-module views of individual, type information, a broad type-agnostic interface, or delegating interpretation of string payloads to callers.
- Leaving variable metadata defined in generator methods to reduce the size of metadata storage down to a 4-byte function pointer. An example of this is provided below:

```
SettingString TimeSource {
        []() {
            return SettingMetadata {
                    "TimeSource",
                    SettingConfig::rwPolicy(),
                    DeviceModel1_6 {"TimeSource"},
                    DeviceModel2_0 {{"ClockCtrlr"}, {"TimeSource"}, {
                            std::nullopt,
                            ocpp2_0::DataEnumType::kSequenceList,
                            std::nullopt,
                            std::nullopt,
                            "Heartbeat"
                    }},
                    "Heartbeat"
            };
        },
        [](auto const& value) {return value == "Heartbeat";}
};
```

OCTT test cases

At the time of writing all core OCPP 2.0.1 test cases marked as required by the PICS tool were passing (with the exception of TC_B_49 which was failing due to a confirmed OCTT tool bug). Note that full compliance with the Smart Charging certification profile was targeted but not fully implemented at the time of writing (and hence has been excluded from the list of test cases below):

Table 8. OCTT test cases

Section	Test Cases
Security	TC_A_01, TC_A_04, TC_A_05, TC_A_06, TC_A_09, TC_A_10, TC_A_19, TC_A_20, TC_A_22
Provisioning	TC_B_01, TC_B_02, TC_B_03, TC_B_06, TC_B_07, TC_B_09, TC_B_10, TC_B_11, TC_B_12, TC_B_13, TC_B_20, TC_B_21, TC_B_22, TC_B_23, TC_B_28, TC_B_29, TC_B_30, TC_B_32, TC_B_33, TC_B_34, TC_B_35, TC_B_36, TC_B_37, TC_B_39, TC_B_43, TC_B_45, TC_B_46, TC_B_49, TC_B_50, TC_B_51, TC_B_52, TC_B_53, TC_B_57
Authorization	TC_C_02, TC_C_04, TC_C_06, TC_C_07, TC_C_26, TC_C_39, TC_C_42, TC_C_56
Transactions	TC_E_01, TC_E_03, TC_E_04, TC_E_05, TC_E_06, TC_E_07, TC_E_09, TC_E_10, TC_E_13, TC_E_15, TC_E_16, TC_E_21, TC_E_28, TC_E_30, TC_E_31, TC_E_32, TC_E_33, TC_E_34, TC_E_35, TC_E_38, TC_E_39, TC_E_40, TC_E_41, TC_E_42, TC_E_43, TC_E_44, TC_E_50, TC_E_51
Remote Control	TC_F_01, TC_F_02, TC_F_03, TC_F_04, TC_F_08, TC_F_09, TC_F_13, TC_F_14, TC_F_17, TC_F_18, TC_F_19, TC_F_20, TC_F_23, TC_F_24, TC_F_26, TC_F_27
Availability	TC_G_01, TC_G_02, TC_G_03, TC_G_04, TC_G_05, TC_G_06, TC_G_07, TC_G_08, TC_G_09, TC_G_10, TC_G_11, TC_G_12, TC_G_13, TC_G_14, TC_G_15, TC_G_16, TC_G_17, TC_G_18, TC_G_19, TC_G_21
Meter Values	TC_J_01, TC_J_02, TC_J_03, TC_J_07, TC_J_08, TC_J_09, TC_J_10
Smart Charging	TC_K_38
Firmware Management	TC_L_01, TC_L_02, TC_L_03, TC_L_05, TC_L_06, TC_L_07, TC_L_08, TC_L_10, TC_L_15, TC_L_18
Certificate Management	TC_M_01, TC_M_02, TC_M_07, TC_M_12, TC_M_13, TC_M_17, TC_M_18, TC_M_19, TC_M_20, TC_M_22
Diagnostics	TC_N_25, TC_N_26, TC_N_32, TC_N_35
Data Transfer	TC_P_01, TC_P_03