

C12.19 Program Sequence Software High Level Design

On-Ramp Wireless Confidential and Proprietary. This document is not to be used, disclosed, or distributed to anyone without express written consent from On-Ramp Wireless. The recipient of this document shall respect the security of this document and maintain the confidentiality of the information it contains. The master copy of this document is stored in electronic format, therefore any hard or soft copy used for distribution purposes must be considered as uncontrolled. Reference should be made to On-Ramp Wireless to obtain the latest revision.

On-Ramp Wireless Incorporated
10920 Via Frontera, Suite 200
San Diego, CA 92127
U.S.A.

Copyright © 2015 On-Ramp Wireless Incorporated.
All Rights Reserved.

The information disclosed in this document is proprietary to On-Ramp Wireless Inc., and is not to be used or disclosed to unauthorized persons without the written consent of On-Ramp Wireless. The recipient of this document shall respect the security of this document and maintain the confidentiality of the information it contains. The master copy of this document is stored in electronic format, therefore any hard or soft copy used for distribution purposes must be considered as uncontrolled. Reference should be made to On-Ramp Wireless to obtain the latest version. By accepting this material the recipient agrees that this material and the information contained therein is to be held in confidence and in trust and will not be used, copied, reproduced in whole or in part, nor its contents revealed in any manner to others without the express written permission of On-Ramp Wireless Incorporated.

On-Ramp Wireless Incorporated reserves the right to make changes to the product(s) or information contained herein without notice. No liability is assumed for any damages arising directly or indirectly by their use or application. The information provided in this document is provided on an “as is” basis.

This document contains On-Ramp Wireless proprietary information and must be shredded when discarded.

This documentation and the software described in it are copyrighted with all rights reserved. This documentation and the software may not be copied, except as otherwise provided in your software license or as expressly permitted in writing by On-Ramp Wireless, Incorporated.

Any sample code herein is provided for your convenience and has not been tested or designed to work on any particular system configuration. It is provided “AS IS” and your use of this sample code, whether as provided or with any modification, is at your own risk. On-Ramp Wireless undertakes no liability or responsibility with respect to the sample code, and disclaims all warranties, express and implied, including without limitation warranties on merchantability, fitness for a specified purpose, and infringement. On-Ramp Wireless reserves all rights in the sample code, and permits use of this sample code only for educational and reference purposes.

This technology and technical data may be subject to U.S. and international export, re-export or transfer (“export”) laws. Diversion contrary to U.S. and international law is strictly prohibited.

Ultra-Link Processing™ is a trademark of On-Ramp Wireless.

Other product and brand names may be trademarks or registered trademarks of their respective owners.

C12.19 Program Sequence Software High Level Design

013-xxxx-00 Rev. B

February 2, 2015

Contents

1 Introduction	1
1.1 Purpose	1
1.2 Scope	1
1.3 References	1
2 C12.19 Program Sequence Overview	3
2.1 Software Design Requirements	3
2.2 Application Overview	4
2.3 Run-Time Environment	7
2.4 C12.19 Program Sequence States	11
2.5 C12.19 Memory Resources	11
2.5.1 Program FLASH	11
2.5.2 SRAM	11
2.5.3 NVM	12
3 C12.19 Program Sequence Components	13
3.1 USP Support for Resumable C12.19 Request	13
3.2 C12.19 Request Header Verification	13
3.3 MCM Application Handlers	14
3.4 AMI Proto Handlers	15
3.5 Meter Layer Support	17
3.6 C12.18 Link Layer Support	18
4 C12.19 Program Sequence External Interfaces	19
4.1 Over-the-Air Interface	19
4.1.1 C12.19 Request Message	19
4.1.2 C12.19 Response Message	23
4.1.3 C12.19 Abort Message	25
4.1.4 AMI Result Definitions	25
4.2 Host Interface	26
4.2.1 C12.19 Request Download Message	27
4.2.2 C12.19 Response Download Message	28
4.2.3 C12.19 Request Upload Message	30
4.2.4 C12.19 Request Generation Utility	31

5 C12.19 Program Sequence Internal Interfaces	34
5.1 HOST_CMN Layer APIs.....	34
5.1.1 HOST_CMN_APP_USP_RxMsgFrag()	34
5.1.2 HOST_CMN_APP_USP_TxMsgFrag()	35
5.1.3 HOST_CMN_APP_USP_ResetRxStream()	35
5.2 MCM APPLICATION Layer APIs	35
5.3 AMI_PROTO_C1219 Layer APIs	36
5.3.1 AMI_PROTO_C1219_Init()	36
5.3.2 AMI_PROTO_C1219_RxResumableStreamHandler()	36
5.3.3 AMI_PROTO_C1219_HandleMcmAppMsgC1219Req()	36
5.3.4 AMI_PROTO_C1219_DecodeOperation()	37
5.3.5 AMI_PROTO_C1219_DecodeTableData()	37
5.3.6 AMI_PROTO_C1219_FlushTableData()	38
5.3.7 AMI_PROTO_C1219_GetRemainingC1219ReqSize()	38
5.3.8 AMI_PROTO_C1219_GetRemainingTableDataSize()	38
5.3.9 AMI_PROTO_C1219_AllocateResponse()	39
5.3.10 AMI_PROTO_C1219_AllocateReadStorage()	39
5.3.11 AMI_PROTO_C1219_PrepOperationRsp()	39
5.3.12 AMI_PROTO_C1219_CommitBulkReadData()	40
5.3.13 AMI_PROTO_C1219_FinishOperationRsp()	40
5.3.14 mcmAppMsgCallbackC1219Req()	41
5.3.15 AMI_PROTO_C1219_AbortMcmAppMsgC1219req()	42
5.3.16 AMI_PROTO_C1219_SendErrorInd()	42
5.4 METER Layer APIs	43
5.4.1 METER_ProgramSequenceReq()	43
5.4.2 METER_CheckProgramSequenceState()	43
5.5 C12.18 APP Layer APIs	43
5.5.1 C12_18_APP_ReadFullTable()	44
5.5.2 C12_18_APP_ReadPartialTable()	44
5.5.3 C12_18_APP_WriteFullTable()	44
Appendix A Abbreviations and Terms	45
Appendix B C12.19 Program Sequence Flow Charts.....	46
B.1 C1219Req Downlink Stream Management	46
B.2 C1219Req Header Validation	47
B.3 MCM Application C1219Req Processing	48
B.4 METER Layer C1219Req Processing	49
B.5 METER Layer C1219 Operation Processing	51
B.6 C1219 READ Action Processing	53
B.7 C1219 WRITE Action Processing.....	56
B.8 C1219 PROCEDURE Action Processing	58

B.9 C1219 PATCH Action Processing	60
B.10 C1219 DELAY Action Processing.....	62
B.11 C1218 LINK Layer Receive Packet Processing	63
B.12 C1218 LINK Layer Transmit Packet Processing	64
B.13 METER Layer C1219Rsp Processing	65
B.14 C1219Rsp Uplink Stream Management	67
Appendix C System Test Outline	68

Figures

Figure 1. C12.19 Program Sequence Overview5

Figure 2. C12.19Req Downlink Run-Time Processing Distribution 8

Figure 3. C12.19Rsp Uplink Run-Time Processing Distribution 10

Figure 4. C12.19 Request Download Example28

Figure 5. C12.19 Response Download Example30

Figure 6. C12.19 Request Upload Example31

Figure 7. C12.19 Request Generation Example33

Tables

No table of figures entries found.

Revision History

Revision	Release Date	Change Description
A	Aug 12, 2014	Initial release.
B	Feb 2, 2015	Updated for bitfield write and read/validate operations.

1 Introduction

1.1 Purpose

This document describes the software implementation and methods used on the ORW EMCM platform used to execute a C12.19 Program Sequence used to interface with a range of On-Ramp Wireless' AMI meter solutions (e.g., Raptor, Ptero, etc).

1.2 Scope

This high level design (HLD) document focuses only on the existing software components within the EMCM code base have been modified to support the C12.19 Program Sequence implementation. As such, it is primarily intended for review by internal developers and system-level designers. It is assumed that the reader of the document has basic familiarity with the C12.19 and C12.18 standards as it relates with the programming and communication with an ANSI Electrical Meter.

However, the actual overall software architecture of the EMCM firmware and board level support (i.e., buffer management, hardware drivers, reset management) is beyond the scope of this document.

In addition, support for the C12.19 Program Sequence on the network-side components (e.g., OTV, HES, the USP transport layer) is also beyond the scope of this document.

1.3 References

The following documents are referenced and provide more detail on various components that need to be considered in the C12.19 Program Sequence design and implementation:

1. *AMI_1.x HLD* - http://jenkins/view/AMI/job/trunk_ami_ota_inf/lastSuccessfulBuild/artifact/ami_ota_inf/doc/output/ami_hld.pdf
Requirements and high-level system design of ORW AMI solutions – the C12.19 Program Sequence is a scheduled feature for the AMI_1.2 release and is presented in section 3.7 of this document.
2. *EMCM Software High Level Design document (xxx-xxxx-xx)*
If such a document actually existed – this design document would certainly reference it since it is leveraging many of the current firmware design features/resources.
3. *Data Object Manager Software High Level Design document (013-xxxx-00)*
Provides details regarding the Data Object Manager software component on EMCM which manages all of the C12.19 Persistent Data Structures in external NVM.
4. *Google Protocol Buffer* – <https://developers.google.com/protocol-buffers/>

Provides details on the encoding format used for C12.19 Program Sequence Request and Response messages.

5. *Nanopb Documentation* - <http://koti.kapsi.fi/jpa/nanopb/>

Provides details regarding the third-party Nanopb software component used on EMCM to encode and decode the Protocol Buffer message formats for C12.19 Program Sequence Request and Response messages.

6. *ANSI Std C12.19-1997*

Utility Industry End Device Data Tables – i.e., a list of the ANSI Meter data tables that the C12.19 Program Sequence is accessing.

7. *ANSI Std C12.19-1996*

Protocol Specification for ANSI Type 2 Optical Port – i.e., details on the protocol and packet format used by the EMCM Meter Layer to communicate with an ANSI Meter.

8. *MCM Program Sequence Ladder Diagrams* -

http://omega.onramp.local/projects/OnRampTrac/browser/docs/trunk/sw/mcm_program_sequence.odg

Open Office Draw source of all C12.19 Program Sequence software flow charts/ladder diagrams found in Appendix B.

2 C12.19 Program Sequence Overview

2.1 Software Design Requirements

The following software design requirements are considered for the C12.19 Program Sequence software implementation:

- Implementation of the OTA C12.19 Program Update sequence as specified in the AMI_1.1 HLD (see reference [1]) which can be summarized as follows:
 - Support a sequential series of C12.19 level accesses/operations that interfaces with an ANSI compliant meter using a serial C12.18 protocol layer (see references [6] and [7]).
 - Message size support for C12.19 Program Update in both the uplink and downlink of up to 50kbytes.
 - C12.19 Program Update uplink/downlink message will leverage On-Ramp's wireless USP resumable stream support (i.e. fragmentation support for message data with persistent stream state).
 - Optional support on-target for a configurable start and/or expiration time for a C12.19 Program Update (as opposed to immediate execution).
 - Optional support for an auto-retry mechanism of a C12.19 Program Update that fails either because of a meter and/or application host exception.
- The C12.19 Program Update utilizes USP Resumable stream support for messages and stream state for both uplink and downlink message exchanges with the network backend network components. This is done through use of the Data Object Manager software component (see reference [3]) with wear-leveling and flash driver support for the following data objects:
 - Tx Stream State – Used by the host_cmn USP handlers for all uplink C12.19 response messages/status.
 - Rx Stream State – Used by the host_cmn USP handlers for all downlink C12.19 Requests management messages.
 - Message Request Buffer– A 50k buffer used as the fragment assembly and staging area for all downlink C12.19 Requests.
 - Message Response Buffer – A 50k buffer used as the fragment assembly and staging area for all uplink C12.19 Responses.
 - Bulk Read scratch buffer – A 20k buffer used as a staging area for any C12.19 bulk read data from the ANSI meter.
- The MCM APP layer will perform initial processing and integrity validation of an incoming C12.19 Request Message using the traditional AMI header (i.e., 2-byte message opcode, and 4-byte CRC calculated over the header/payload data).

- The MCM AMI PROTO layer will support Google protocol buffer encoding and decoding for all C12.19 operations through use of the third-party nanopb software library. Because the size of a C12.19 Request/Response can be quite large, nanopb encoding/decoding is done in manageable stream “chunks”.
- The METER Layer will process the C12.19 Program Sequence to completion before servicing any subsequent METER requests from the application – this includes any scheduled table reads that may occur during a network schedule Program Upgrade operation.
- The METER Layer will process the C12.19 Program Sequence in a re-entrant manner so that the EMCM application can allow other processes a chance to run between C12.19 level accesses with the ANSI Meter.
- Where possible, the C12.19 Program Sequence implementation will reuse existing EMCM software resources/routines in order to keep both additional code space and RAM usage as minimal as possible.

Listed are the assumptions and limitations of the proposed implementation:

- The C12.19 Program Sequence is implemented as specified in the C1219 Request message – it is the responsibility of the user and the head-end to understand the limitations of the ANSI Electrical Meter with regards to table data, table data access procedures, as well as any transfer size restriction over the C12.18 protocol (e.g., will a FULL table data read be accommodated by the C12.18 negotiation layer). The EMCM will make no attempt to modify/interpret C12.19-level accesses other than what is specified.
- There is only one C12.19 Program Sequence active for any given endpoint in the AMI network – if a C12.19 Request is received while an existing C12.19 Request is pending, the new request will cancel/override the existing request.
- The C12.19 Program Sequence can block servicing the METER layer event queue for a long duration until the program sequence runs to completion (including any retries). The EMCM application will need to be tolerant of this lock-out period by the METER layer.

2.2 Application Overview

The C12.19 Program Sequence can be view quite simply as a set of C12.19 level accesses to an ANSI meter that is executed in a specified order. It is meant as over-the-air (OTA) proxy for third-party ANSI C programming tools that normally are used at the site of an installed meter through use of its C12.18 Optical Port. However, because this feature utilizes ORWs’ RPMA network capabilities, the C12.19 Program Sequence can be scheduled remotely as well as to groups of AMI endpoints using USP Multi-User Downlink (MUD) capabilities. Further details regarding the high-level AMI requirements of the C12.19 Program Sequence can be found in section 3.7 in the ORW AMI1.1 HLD (see reference [1]).

The implementation of the C12.19 Program Sequence in the EMCM firmware is mainly concerned with the following functional elements:

- Fragment assembly and validation of a C12.19 Request message from the network using USP into a staging area in external NVM.
- Scheduling when to execute a staged C12.19 Request.

- The ordered/procedural execution of C12.19 operations from a staged C12.19 Request.
- The ordered/procedural response handling from an ANSI meter for each C12.19 Operation stored into a separate staging area in external NVM.
- Fragment handling and stream state management for the C12.19 Response sent back to the network from its staging area in external NVM.
- Exception handling and retry mechanism for a C12.19 Program Sequence that may occur for both Meter and Host Application failures.

The following diagram illustrates the basic application overview and flow of data for the C12.19 Request/Response messages on an EMCM endpoint:

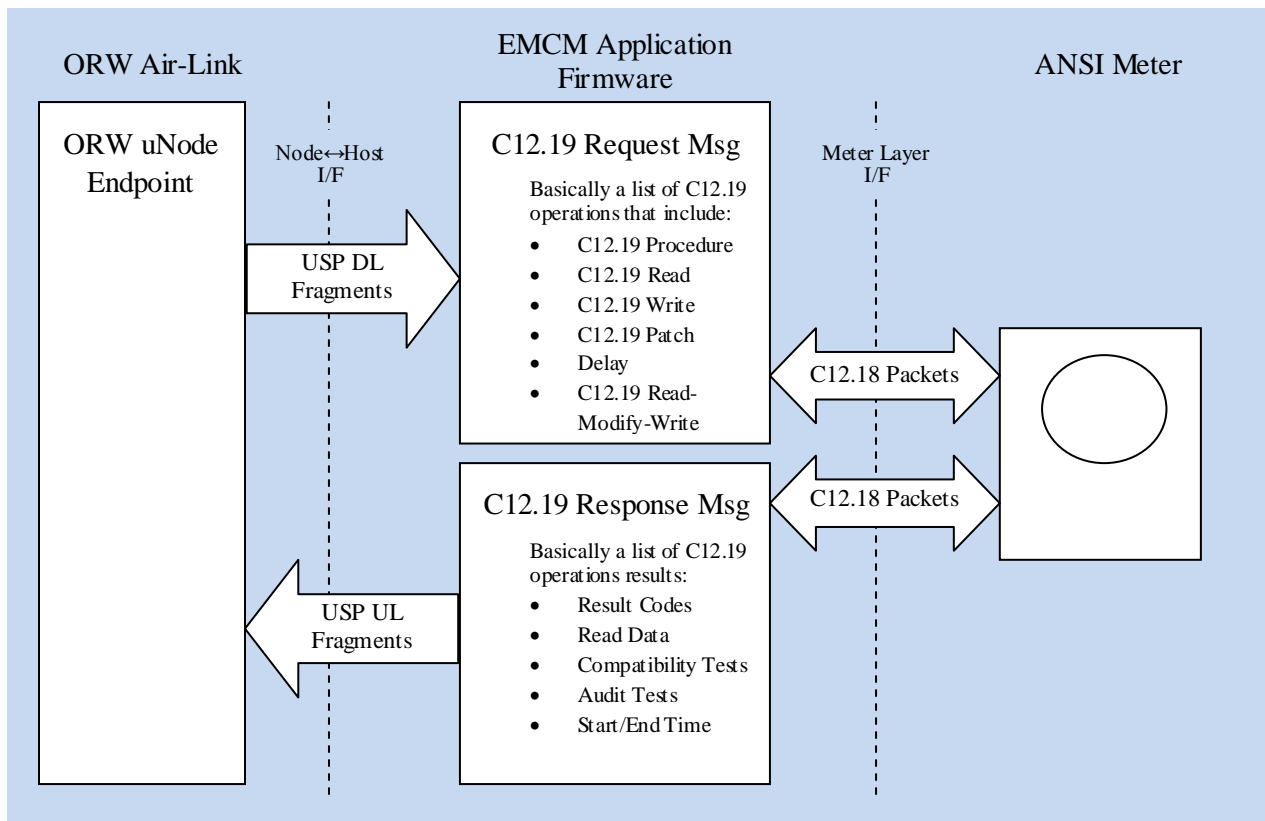


Figure 1. C12.19 Program Sequence Overview

C12.19 Operations/Actions are classified into the following (as described in section 3.7 of reference [1]):

- **C12.19 Read Action:** Used to access and report data for a specified C12.19 Tables on an ANSI compliant meter. The table read can either be FULL or PARTIAL – however, a PARTIAL read requires a length and offset parameter to be defined. NOTE: FULL reads on ANSI GE meters will only read up to the negotiated C12.18 limit on read transfers – this may not be enough to report the entire table data – it is preferred that C12.19 Read Actions use the PARTIAL read method.

A C12.19 Read action *must* have a table number and table type (i.e., Standard or Manufacturing) defined or it will be treated as a malformed C12.19 Operation by the

MCM firmware. Further details on C12.19 Tables can be found either in the C12.19 standard (see reference [7]) or the data sheet for a particular vendor meter model – the former is preferred as it will list the manufacturing tables that are specific to the meter.

C12.19 Read actions can be further classified by an action subtype:

- The standard Read-only subtype (default) – read data is simply reported over the C1219Rsp message.
- A COMPATIBILITY subtype – the first set of READ actions in a C12.19 Program Sequence that ensures the ANSI meter is the correct targeted endpoint by comparing read data with a set of defined validation data. A failed COMPATIBILITY check will terminate the C12.19 Program Sequence regardless of any specified retry attempts.
- An AUDIT subtype – the last set of READ actions in a C12.19 Program Sequence that checks that the ANSI meter has been correctly programmed by verifying read data with a set of defined validation data. Unlike, COMPATIBILITY subtypes, retries are permitted as a result of a failed AUDIT check.

As noted, COMPATIBILITY and AUDIT subtypes will include additional message descriptors that define the validation data and the method in which it will be applied – e.g., hex data compare, integer compare, range check, etc.

- **C12.19 Write Action:** Used to update data in a specified C12.19 Tables on an ANSI compliant meter. The table write can also be either FULL or PARTIAL – however, a PARTIAL write requires both a length and offset parameter to be defined while a FULL write always starts at offset zero.

As with reads, A C12.19 Write action *must* have a table number, table type (i.e., Standard or Manufacturing), and a non-zero length table data array defined or it will be treated as a malformed C12.19 Operation by the MCM firmware. Further details on C12.19 Tables can be found either in the C12.19 standard (see reference [7]) or the data sheet for a particular vendor meter model – the former is preferred as it will list the manufacturing tables that are specific to the meter.

- **C12.19 Procedure Action:** ANSI Procedures are a specialized meter operations programmed through ANSI Standard Table 7. Similar to the C12.19 Table data, there are both Standard and Manufacturing Procedures – both of which may include user-defined table data (similar to a Table Write).

A C12.19 Procedure *must* have a table number and table type (i.e., Standard or Manufacturing) or it will be treated as a malformed C12.19 Operation by the MCM firmware.

- **C12.19 Patch Action:** This is a special C12.19 Program Sequence action not defined by the ANSI standard. Rather, it is a method to replace a section of table write data with run-time determined data for the *next* C12.19 Operation in the program sequence table – currently, the C12.19 Patch action is only valid if it precedes a WRITE or PROCEDURE action (otherwise the patch data is lost).

Currently, the following patch types are supported on the EMCM firmware:

- Local Time Patch – essentially the local calendar time is retrieved from the ANSI meter and replaces a portion of the next C12.19 operation table data. The byte format of the local time patch is represented as:

Byte[0] – Year – 2000

Byte[1] – Month

Byte[2] – Day

Byte[3] – Hour

Byte[4] – Minute

Byte[5] – Second

A C12.19 Patch action must have a Patch Type, offset, and length defined or it will be treated as a malformed C12.19 Operation by the MCM firmware. In addition, if the patch length exceeds the patch type capability (e.g., more than 6 bytes for Local Time Patch), it will also be flagged as an invalid operation.

- **Delay:** This is simply a straightforward delay, executed on the EMCM host application, applied before the next C12.19 Operation is parsed and executed.
- **Read-Modify-Bitfield-Write:** A two-step READ/WRITE action where only a designated bitfield is updated with a specified value – both C12.19 level table accesses are always done as a using the C12.18 PARTIAL method (i.e., a 1-byte access at a given table offset).

All of the fields associated with the C12.19 Operations are defined in the OTA Protobuf message (see section 4.2.1).

2.3 Run-Time Environment

The C12.19 Program Sequence implementation is but one supported feature in On-Ramps' AMI EMCM application (see reference [2]). The EMCM application, at least on AMI1.1, does not utilize an RTOS – instead, it is implemented in with a single thread of execution which invokes a number of run-time executables in a round-robin fashion to allow each a chance to execute its processes. It should be noted that if one of the MCM's run-time executables take too long to execute one its processes, it will essentially block all other run-time executables until it completes – this implies a bit of trust/discipline on each of the MCM's components to avoid this type of run-time congestion.

The C12.19 Program Sequence is distributed amongst a number of these run-time executables along with a large number of utility functions that can be invoked from the different layers – specifically the Data Object Manager (DOM) and AMI_PROTO decoding/encoding routines.

The following diagram helps illustrate the distribution of the Downlink (DL) C12.19 Request Program Sequencing amongst the afore-mentioned run-time executables:

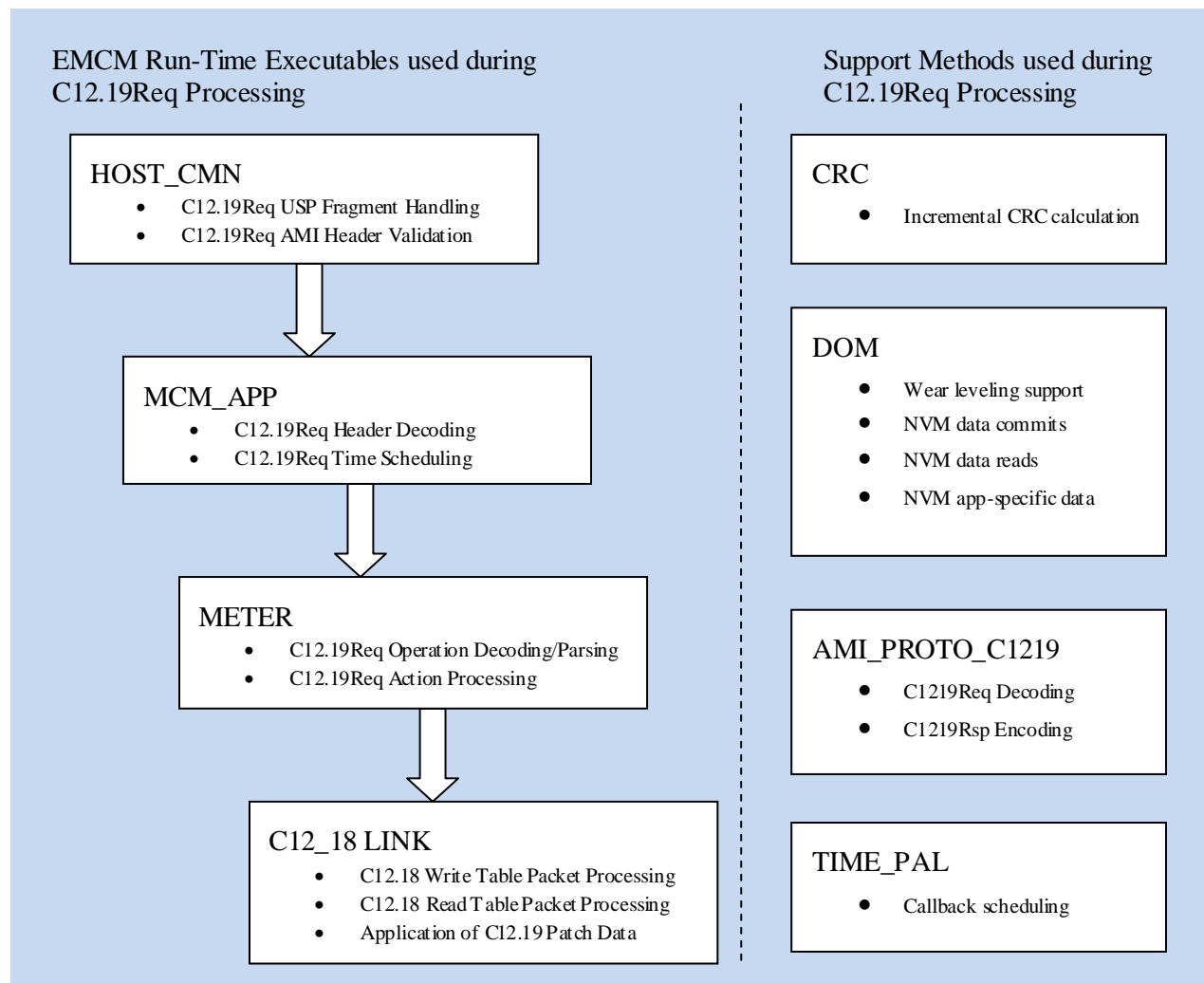


Figure 2. C12.19Req Downlink Run-Time Processing Distribution

The run-time executables involved in Downlink C12.19 Program Sequence, listed from network-level to meter-level processing are as follows:

- **HOST_CMN:** The Host Common software component is responsible for handling each Downlink C1219Req fragment through use of the ULP Stream Protocol (USP). When a valid fragment is received over the dedicated C1219 stream (stream10), it will, in turn, use the Data Object Manager to commit each fragment into external Flash. When the total C1219Req is received, it will validate the data against the received AMI Header.
- **MCM_APP:** The MCM Application handler handles initial decoding of the C1219Req message header data (i.e., all C1219Req fields that precede the list of C1219 Operations) and, if necessary, any scheduling of the C1219Req message. In addition, the MCM_APP, when coming out of reset, will also check/re-schedule any pending C1219Req events.
- **METER:** The METER layer is the “brains” of the C12.19 Program Sequence implementation and is responsible for the parsing and execution of all C1219 Operations to the actual ANSI Meter.

- **C12_18_LINK:** The C12_18 Link layer manages each serial packet to/from the ANSI meter. During C1219 Operation execution for either WRITE or PROCEDURE actions, it will be necessary to callback/access the C1219Req buffer for user-specified table data bytes.

Supporting the listed run-time executables are the following software components:

- **CRC:** The 32-bit Cydic Redundancy Check calculation functions are used in EMCM for verification of all AMI OTA messages (including the C1219Req and C1219Rsp).
- **DOM:** The Data Object Manager is the primary interface between the C12.19 Program Sequence and all accesses to the message buffers and bulk read data in external NVM – the DOM fragment handlers are used extensively during encoding and decoding of the protobuf formatted data stream. Details on the Data Object Manager can be found in reference[3].
- **AMI_PROTO_C1219:** Advanced Meter Infrastructure (AMI) protocol utility functions are used for the decoding/encoding of all AMI messages with support from the third party nanopb software component (see reference [5]). The C12.19 Program Sequence implementation is slightly different in that it manually decodes/encodes messages in partial “chunks” due to RAM buffer constraints in the EMCM firmware.
- **TIME_PAL:** The Timer Platform Abstraction Layer (PAL) provides the necessary driver support to allow the C12.19 Program Sequence to schedule callback events (e.g., delayed start, DELAY actions, etc). Details on the TIME_PAL software component are beyond the scope of the document.

The following diagram helps illustrate the distribution of the Uplink (UL) C12.19 Request Program Sequencing amongst the EMCM run-time executables:

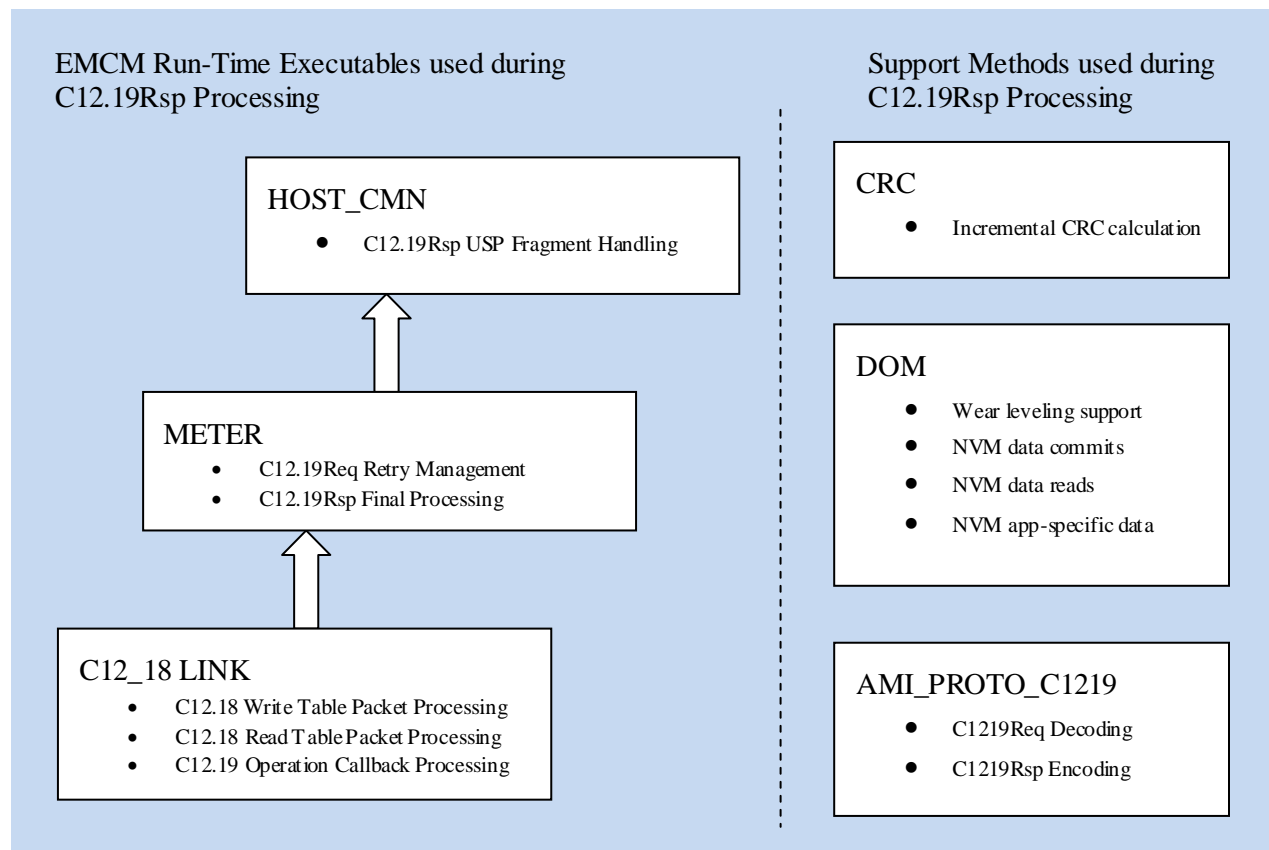


Figure 3. C12.19Rsp Uplink Run-Time Processing Distribution

The run-time executables involved in Uplink C12.19 Program Sequence, listed from meter-level to network-level processing are as follows:

- **C12_18 LINK:** As part of downlink processing, the C12_18 Link layer manages each serial packet to/from the ANSI meter. However, for READ actions, callbacks at the packet layer will be invoked to update the C1219Rsp buffer for any read data bytes. Also, once the C12.18 session is completed, user-callbacks at the end of the transaction will perform the final C1219 operation response encoding.
- **METER:** At the conclusion of a successful C12.19 Program Sequence, the METER layer will be responsible for encoding the final result code, or in the case that a C12.19 level access resulted in a failure, it will re-enqueue the same sequence for any qualified retry attempt.
- **HOST_CMN:** As with the downlink path, the Host Common software component is responsible for handling each Uplink C1219Rsp fragment through use of the ULP Stream Protocol (USP). This too uses the Data Object Manager to retrieve each fragment from external Flash when requested by the ORWNode.

The AMI support blocks are still used in the same manner as in Downlink processing.

It is important to note that none of the C12.19 Program Sequence software methods are used in interrupt context.

2.4 C12.19 Program Sequence States

For the most part, the C12.19 Program Sequence is an event driven implementation within the MCM firmware framework.

However, at the METER layer, there is a new action state defined within by the *MeterTypeOfAction_t*:

- **METER_ACTION_PROGRAM_SEQUENCE** – The METER layer has received a Meter Program Sequence request through its event queue and is in the process of executing the C12.19 Program Sequence.

While the METER_ACTION_PROGRAM_SEQUENCE Action state is active the METER will:

- Block all processing of queued user action requests in the user Action Request Queue in the METER layer run-time executable.
- Processing of the METER layer event flags are handled normally. In fact, the PROGRAM_SEQUENCE event flag is the primary mechanism of triggering C12.19-level accesses while the METER_ACTION_PROGRAM_SEQUENCE Action state is set.

The METER layer will only exit the METER_ACTION_PROGRAM_SEQUENCE Action state once the C12.19 Program Sequence completes its final processing step.

NOTE: The METER-layer run-time executable can be blocked from servicing the user Action Request Queue for a considerable amount of time while a C12.19 Program Sequence is being serviced (e.g., 8 seconds for an 80-step program upgrade sequence). It is the responsibility of the MCM user application to effectively manage delayed user requests and/or full event queue due to these types of servicing delays.

2.5 C12.19 Memory Resources

As specified in the overall software design requirements, the C12.19 Program Sequence implementation is designed to reuse memory resources and/or utility functions as much as it is possible – the following subsections identify the memory resources that are added to the MCM firmware base as a result of its implementation:

2.5.1 Program FLASH

The C12.19 Program Sequence currently uses the following program flash space for its implementation:

- **Code Space:** ~2.2Kbytes
- **Read-Only and Initialized Data:** Negligible/None

NOTE: This program flash usage does not take into consideration the NVM DOM/PAL drivers nor any library routines it may call (e.g., memory copy function calls).

2.5.2 SRAM

The C12.19 implementation current uses on-target RAM space for:

- **Data Buffers and tracking variables:** ~500bytes

2.5.3 NVM

The C12.19 Program Sequence implementation leverages external SPI Flash for much of its implementation. The following memory blocks are requested from the Data Object Manager (DOM):

- ULP Stream Protocol (USP) Tx Stream State – 50 x 256byte slots in NVM to provide the necessary wear leveling of Tx Stream State over the lifetime of the EMCM product.
- ULP Stream Protocol (USP) Rx Stream State – 50 x 1.2Kbyte slots in NVM to provide the necessary wear leveling of Rx Stream State over the lifetime of the EMCM product.
- Downlink C12.19 Message Request Buffer – 5 x 50k slots in NVM to provide the necessary wear leveling of C12.19 Requests over the lifetime of the EMCM product.
- Uplink C12.19 Message Response Buffer – 5 x 50k slots in NVM to provide the necessary wear leveling of C12.19 Responses over the lifetime of the EMCM product.
- Meter Bulk Read Scratch Buffer – 5x20k slots in NVM to provide the necessary wear leveling of the bulk read storage area (used in support of a C12.19 Program Sequence).

In sum, approximately 672kbytes of external Flash is used for the underlying C12.19 Program Sequence Implementation – well within the 2MB/8MB flash parts currently used for all current ORW EMCM platforms.

3 C12.19 Program Sequence Components

The following sections summarize the MCM software changes needed to implement the C12.19 Program Sequence feature – rather than going into intense details on each code changes, the following subsections will make frequent links to the Sequence Ladder diagrams in Appendix B to help illustrate the code changes and interactions between the various MCM software layers and/or run-time executables.

3.1 USP Support for Resumable C12.19 Request

All C12.19 DL and UL OTA messages are configured by both HES and the EMCM firmware as USP resumable transfers over a dedicated USP stream – in this case Stream ID10. Resumable support requires that the both the DL/UL message data and stream state be store in persistent memory – on the EMCM platform, this means these data objects are stored in external SPI Flash.

The C12.19 Program Sequence implementation on MCM leverages the Data Object Manager (or DOM – see reference [3]) for all NVM accesses to these aforementioned data objects (which have been previously listed as NVM resources in section 2.5.3).

The **C1219Req Downlink Stream Management** sequence diagram (see section B.1) illustrates all interactions between the HOST_CMN USP data fragment handlers and the C1219Req buffer in external NVM.

The **C1219Rsp Uplink Stream Management** sequence diagram (see section B.14) illustrates all interactions between the HOST_CMN uplink data fragment handlers and the C1219Rsp buffer.

Because the message and stream data is persistent, stream transfers will resume/complete even after MCM system resets (e.g., power-fail scenarios).

In compliance with the AMI HLD, the HES network component can abort an in-progress transfer of the C1219Req by using the USP Flush mechanism – the MCM firmware, in turn, will flush/dear its buffer in external NVM for next DL stream transfer.

Upon completion of a transfer over DL resumable stream ID#10, the HOST_CMN Rx USP handlers will trigger a call to AMI_PROTO for AMI Header Qualification (see next section). It will also update the App-specific meta-data associated with the C1219Req buffer with the total length of the received data to aid in exception management and/or scheduled retry attempts.

3.2 C12.19 Request Header Verification

AMI Header verification on a DL message received over the dedicated resumable stream for C12.19 Requests (or a C12.19 Request Abort) is the next step in DL processing in an API provided by the AMI_PROTO Layer (see section 5.3.1). This process involves calculating a CRC32 calculation over the received message and comparing it with a CRC32 value in the AMI Header.

However, unlike AMI messages received over the non-resumable DL stream channels, the C1219Req message can be quite large (i.e., up to a 50kbyte max) and needs to be done in

incremental “chunks” extracted from external NVM. This is done using a temporary 256-byte buffer declared on the stack.

The **C1219Req Header Validation** sequence diagram (see section B.2) illustrates all interactions between the AMI_PROTO Verification handler and the C1219Req buffer in external NVM as well as the EMCM CRC calculation module. It should be noted that this verification process is still done in the context of the HOST_CMN run-time executable.

If the CRC is good, the AMI_PROTO verification handler will then enqueue an Rx Message handler event with the MCM_APP Layer (see next section).

If the CRC is bad, the AMI_PROTO verification handler will generate an uplink error message to the network components to help indicate that a malformed C1219Req was received for this endpoint – it will also invoke the Data Object Manager to re-allocate a fresh C1219Req buffer for any subsequent Stream10 DL processing.

3.3 MCM Application Handlers

Receive AMI Messages in the MCM firmware have traditionally been managed via a “LATER” event queue – this queue will only be served by the MCM_APP Layer with transmit message buffers are available to generate the appropriate OTA response (see reference [2]). If the transmit message buffer is not available, processing of the “LATER” event queue will be paused until a buffer resource becomes available (which could be some length of time depending on the amount of network traffic and channel conditions).

The AMI Messages received over the dedicated C1219 resumable stream can be treated in a different manner as its response buffer is already reserved as one of the allocated Data Objects in external NVM. As a result, these AMI received events can be handled in the MCM_APP “NOW” event queue which can be serviced during the next pass through the MCM_APP’s run-time executable.

If the MCM_APP layer will drop the C1219Req if it does has not transitioned beyond the WAITING_FOR_TIME_SYNC state (it needs time sync to get updated POSIX time from the network) or within the POWER_FAIL state. It should be noted that as the MCM_APP transitions out of the WAITING_FOR_TIME_SYNC state, it will autonomously check for any pending C1219Req messages in external Flash and, if detected, will (re)validate/enqueue the request for processing.

Once the C1219Req “NOW” event is processed by the MCM_APP run-time executable, it will, with the aid of the nanopb decoding function APIs (see next section), decode/save all C1219Req message tags/variables which precede the array of C1219 Operations. The extracted startTime and/or expiryTime (or lack thereof), will determined if the whether or not the C1219Req is to be processed immediately or scheduled in the EMCM firmware as a future event (via TIMER support in software).

The **MCM Application C1219Req Processing** sequence diagram (see section B.3) illustrates all interactions between the MCM_APP C1219Req handler and the AMI_PROTO layer for initial decoding and scheduling of the C12.19 Program Sequence.

If C1219Req needs to be executed, a C1219 Program Sequence event is enqueued with the METER layer. It should be noted that once this event is enqueued, Rx Reliable stream processing

will need to be paused to prevent overwrite of the C1219Req buffer in external NVM – corruption of a C1219Req while its being processed by the METER layer can lead to unpredictable run-time processing on the EMCM and/or ANSI Meter and must be avoided.

3.4 AMI Proto Handlers

The AMI_PROTO layer can be considered the “brains” of the C12.19 Program Execution implementation on the EMCM firmware. It provides the utility functions for all of the C1219Req Decoding steps, all of the C1219Rsp Encoding steps, performs any necessary read validation processing, and is responsible for maintaining the tracking variables used stepping through the procedural list of C12.19 level accesses.

All encoding and decoding routines heavily leverage the third party software utility, *nanopb* (see reference [5]), to manually encode/decode the C1219 Request and Response message contents. However, unlike the traditional AMI message decode/encode support, AMI_PROTO support for the C12.19 Program Execution heavily leverages the manual decoding methods – this is required because of the potentially large size of a C1219Req/Rsp message (up to 50Kbytes) which can greatly exceed the AMI_PROTO's decode buffer (currently sized as 1Kbyte). Manual decoding methods are done on partial message “chunks” loaded into RAM, initialized as nanopb stream object, and using the aforementioned decoding methods for one of the three wire types used in C1219 program sequence processing:

- VARINT wire types (bit-packed data fields, up to 64-bit size data, following a tag element).
- FIXED wire types (32-bit sized fixed data following a tag element)
- STRING write types – a packed byte array *or* nanopb submessage depending on the tag element – A first VARINT field following a STRING wire type is the length of the encoded array/submessage.

Details on the *nanopb* implementation and coding methods are beyond the scope of this document – however, to properly understand how AMI_PROTO manages message decoding/encoded, it is recommended that *nanopb* stream management methods be reviewed to see how it ties into the AMI_PROTO handlers.

AMI_PROTO utility APIs are included for the following decoding/encoding methods:

- Initial C1219Req parsing (to support MCM_APP initial processing as discussed in the previous section – see section 5.3.3 for a description of the API).
- Meter Layer C1219Req parsing (to support METER layer Program Sequence event handler – see section 5.3.4 for a description of the API).
- C1219 Operation submessage parsing (see section 5.3.4)
- C1219 Table data parsing (see section 5.3.5)
- C1219 Table data flush (for exception management – see section 5.3.6)
- C1219 Response prep (see section 5.3.11)
- C1219 Read data commits (see section 5.3.12)

- C1219 Operation response encoding (see section 5.3.13)
- C1219 Response final encoding and dispatch (see section 5.3.14)

All of the aforementioned utility APIs rely on tracking variables in the AMI_PROTO control block to properly determine the offset and length of the various coding operation to/from their respective message buffers in external NVM. These tracking variables are initialized during the initial METER Layer C1219Req processing and are updated by each encode/decode call into the AMI_PROTO.

The list of these tracking variables is as follows:

- *totalStreamSize* – The total size of the C1219Req buffer as initially determined by the USP Rx Fragment handlers for the last received C1219Req fragment. This value is also saved as part of the C1219Req application meta-data field to support exception retries.
- *currentStreamOffset* – The current C1219Req offset where unparsed message data will next be decoded from. This is first initialized to an offset directly following the AMI Header and is updated by each iteration/call of the AMI_PROTO decode utility functions.
- *tableDataSize* – The current size of an unparsed table data byte array whenever a WRITE or PROCEDURE C12.19 Operation submessage is first decoded.
- *currentTableDataOffset* – The current offset into a table data byte array that is unparsed. Initially set to zero when a C12.19 Operation submessage is first decoded and updated by each iteration/call of the AMI_PROTO table data decode utility function.
- *validationDataSize* – The current size of an unparsed C12.19 Validation sub-message optionally included as part of a READ C12.19 Operation (i.e., for COMPATIBILITY or AUDIT actionTypes).
- *rspMsgOffset* – The current C1219Rsp offset where the next encoded C1219Rsp element will be written to. This first initialized to zero and updated by each iteration/call of the AMI_PROTO encode utility functions.
- *bulkReadOffset* – The current Bulk Read buffer offset to indicated the start of a read byte array is located. Updated at the start of a READ or PATCH C12.19 Operation.
- *bulkReadSize* – The current size of an updated read byte array located in the Bulk Read buffer. Set to zero at the start of a READ or PATCH C12.19 Operation and updated by each iteration/call to the AMI_PROTO bulk read update utility function.

The AMI_PROTO Layer provides several accessor functions for use by the METER Layer to help coordinate C12.19 Operation execution with the aforementioned tracking variables:

- An accessor API that reports the remaining/unparsed C1219Req message size (see section 5.3.7).
- An accessor API that reports the remaining C1219Req table data size for the current C12.19 Operation (see section 5.3.8).

3.5 Meter Layer Support

The METER Layer is the run-time executable where the C12.19 Program Sequence is executed from. In coordination with the AMI_PROTO utility functions, it will validate and execute each C1219 Operation in sequential order. While a C12.19 Program Sequence is in progress, the METER Layer event queue will be blocked.

The **METER Layer C1219Req Processing** sequence diagram (see section B.4) illustrates how the MCM_APP enqueues the PROGRAM SEQUENCE event to the METER Layer as well as the initial preparation steps to prepare the C12.19 Response message and bulk read data buffers in external NVM.

The METER will start processing the C12.19 Program Sequence one C12.19 Operation at a time as long as there are remaining C1219Req message bytes to parse (as tracked by the AMI_PROTO tracking variables). The METER Layer, using the AMI_PROTO parsing utilities, will parse/validate a C12.19 Operation and, if necessary, schedule the corresponding ANSI Meter link exchange with the C12_18 APP Layer before releasing its context. Callback handlers for the ANSI Meter link exchange will trigger an event flag back to the METER layer to resume processing. If for any reason the operation failed (e.g., malformed operation, C12.18 link layer error), then the current iteration of the C12.19 Program Sequence will be marked as a failure.

The **METER Layer C1219 Operation Processing** sequence diagram (see section B.5) illustrates how the METER Layer processes each individual C12.19 Operation. Depending on the parsed action, it will then branch into the following METER Layer program-specific processing steps:

- **READ action** – The **C1219 READ Action Processing** sequence diagram (see section B.6) illustrates how the READ action is set-up using the existing METER Layer “singleton” read requests. A read can either be FULL or PARTIAL (albeit FULL reads are limited to a data transfer as negotiated by the C12.18 Layer).
- **WRITE action** – The **C1219 WRITE Action Processing** sequence diagram (see section B.7) illustrates how the WRITE action is set-up using the existing METER Layer “singleton” write request. As with reads, a Table Write request can either be FULL or PARTIAL.
- **PROCEDURE action** – The **C1219 PROCEDURE Action Processing** sequence diagram (see section B.8) illustrates how the generic PROCEDURE action is set-up using the existing METER PROCEDURE “singleton” request. At the C12.18 Link Layer, a PROCEDURE action is simply another form of a Table Write request (i.e., a write update to ANSI Standard Table 7).
- **PATCH action** – The **C1219 PATCH Action Processing** sequence diagram (see section B.9) illustrates a LOCAL TIME patch involving a METER Layer local time request through a series of ANSI Meter Table Reads. The results of the patch operation are then temporarily stored in the Bulk Read buffer in external NVM that will get applied in the next C12.19 Operation (as opposed to getting reported in the C1219Rsp).
- **DELAY action** – The **C1219 DELAY Action Processing** sequence diagram (see section B.10) illustrates how the DELAY action is executed in the MCM firmware. Unlike the other C12.19 Operations, the delay is executed on-target without a C12.18 Link Layer exchange with the ANSI meter.

- **READ_MODIFY_BITFIELD_WRITE action** – Basically, a two-step combination of a READ action (see section B.6) of a single table byte where a designated bitfield is modified to a specified bitfield value (leaving the other bits unmodified). Once set, the table byte is updated on the METER via a follow-up WRITE action (see section B.7). As implemented, this action presumes the use of PARTIAL table accesses (i.e., a designated offset with a byte length set to one).

As stated before, once the C12.19 Operation is complete, the **METER Layer C1219Rsp Processing** sequence diagram (see section B.13) illustrates the follow-up processing to build the corresponding C1219 Operation response submessage in external NVM and, if necessary, perform any follow-up validation checks.

3.6 C12.18 Link Layer Support

The C12.18 APP and LINK Layers are responsible for managing the actual data exchange between the MCM host application and the ANSI Meter – the packet format for all meter data exchanges is essentially unchanged for the C12.19 Program Sequence implementation.

However, the method of transferring user-data (i.e., table read or table write byte arrays) has been enhanced to include an optional user-specified callback that bypasses the Tx/Rx payload buffers used in the C12.18 LINK Layer – this approach is needed due to the potential for bulk table data accesses that far exceed the current RAM buffer allocation (currently at 1Kbyte for read buffers and 256bytes for write buffers).

The METER Layer sets these callbacks during the scheduling of C12.18 Read or Write data exchanges as part of its C12.19 Operation processing (see previous section). The callbacks themselves are AMI_PROTO utility functions which manage the transfer of table data in/out of external NVM (see section 3.4).

The **C1218 LINK Layer Receive Packet Processing** sequence diagram (see section B.11) illustrates the interaction between receive packet processing at the C12.18 LINK Layer and the AMI_PROTO bulk read support utility function – as each incoming C12.18 packet is processed, the user-data is extracted and written to external Flash. When the C12.18 session is ended, callback management will bypass the tail-end transfer of user-data from the C12.18 RAM buffers (as was traditionally done).

Support in the C12.18 LINK Layer for transmit packets is a bit more involved – in addition to the transfer of data as each C12.18 packet is formed, an incremental CRC will need to be calculated over each transmitted packet and updated as the last packet is formed. In addition, the write data may be altered by C12.19 PATCH data, but this is all handled in the AMI_PROTO table write data utility function.

The **C1218 LINK Layer Transmit Packet Processing** (see section B.12) illustrates the interaction between C12.18 Transmit packet formation and the AMI_PROTO table write utility function.

4 C12.19 Program Sequence External Interfaces

4.1 Over-the-Air Interface

The OTA interface protocol is defined by the MAC specification which can be found at the following link:

http://alpha/projects/temponRampbugVer1/wiki/Systems%20Engineering/Release_1_2/MAC

The following messages are defined for use by the C1219 Program Sequence execution on EMCM:

- C12.19 Request Message – A downlink message generated by the network components that contains a sequential list of C12.19 level accesses.
- C12.19 Response Message – The uplink message generated by EMCM as a result of processing the C12.19 Request Message
- C12.19 Abort Message – A downlink message generated by the network components to flush/abort a pending C12.19 Request Message.

These messages are defined using Google’s Protocol Buffer encoding format (see reference [4]) and are sent using the On-Ramps’ Stream Protocol transport layer.

NOTE: Although the ordering of the tag IDs is suggested by the Google Protobuf definitions, it is essential that the ordering be preserved for the C1219 OTA messages due to the on-target manual parsing – this is especially true for the validation and table data – both of which are defined as PB_WT_STRING types. Implementation of the MCM AMI_PROTO methods require that byte array data be parsed as the *last* tag element because they are extracted as raw byte data.

All C12.19 OTA message are sent over the dedicated resumable USP stream (stream ID#10) for both uplink and downlink traffic.

There are also “backdoor” C1219 exceptions reported over the MCM Error Indicators – however, these are only used when the Data Object Manager and/or NVM PAL fail to access external Flash during the C12.19 Request/Response processing - the uplink Error Indicators utilize internal RAM buffers and are sent over the non-resumable low-priority stream (i.e., UL Stream#8).

4.1.1 C12.19 Request Message

The C1219Req Message, using the C12_19_REQ AMI message ID, is defined by the following proto definition:

```
message C1219Req
{
    // The correlation ID for the request.
    optional uint32 corrId = 1;

    // Posix start time to execute C1219Req - if not set, start immediately
    optional fixed32 startTime = 2;
```

```

// Posix expire time to halt any C1219 Req processing - if not set, never expire
optional fixed32 expiryTime = 3;

// Number of times to attempt C1219Req programming - if not set, no retries
optional uint32 numRetries = 4;

// The response verbosity of the C1219Rsp - if not set, the MCM will use the
// C1219_RSP_VERBOSITY_TYPE_FAILURE_DETAILS setting when it generates the C1219Req
// message.
optional C1219RspVerbosityType rspVerbosityType = 5;

// The retry delay in between C1219Req program attempts - if not set, the MCM will use
// a default delay value of 15seconds between attempts.
optional uint32 retryDelay = 6;

message C1219Operation
{
    // The C12.19-level action to perform.
    optional C1219Action action = 1;

    // The actionType to further refine how the program operation is processed - if
    not
    // defined, the default is a standard PROGRAM_AND_READ action type.
    optional C1219ActionType actionType = 2;

    // The patch type - only applicable to PATCH operations.
    optional C1219PatchType patchType = 3;

    // The type of table to act on.
    optional ami.defs.MeterTableType type = 4;

    // The table number to act on for READ_*, WRITE_*, PROCEDURE actions.
    optional uint32 number = 5;

    // For READ_* and WRITE_* actions, explicitly define if it is a FULL or PARTIAL
    action
    optional bool isFull = 6;

    // The offset to read from or write to for READ_PARTIAL, WRITE_PARTIAL, or PATCH
    actions.
    optional uint32 offset = 7;

    // For READ_FULL action this is the maximum data to respond with.
    //
    // For READ_PARTIAL action this is the size of the partial read.
    //
    // For a PATCH action, this is the size of the patch data to replace with run-time
    C12.19 data
    optional uint32 maxLength = 8;

    // A delay value, in Msec, for the DELAY action
    optional uint32 delayMsec = 9;

    // A hex bitfield mask for the READ_MODIFY_BITFIELD_WRITE action - although
    declared in protobuf
    // as a variable-sized uint32, only 8-bit hex values are accepted by target
    firmware.
    optional uint32 hexBitFieldMask = 10;

    // A hex bitfield value for the READ_MODIFY_BITFIELD_WRITE action that will
    replace the bits
    // specified in the hexBitFieldMask tags - obviously, the value must be in the
    specified range.
    // ONLY the '1'-bits set in the hexBitFieldMask are updated with this this field.
    optional uint32 hexBitFieldValue = 11;

    // Included for WRITE_FULL, WRITE_PARTIAL, PROCEDURE actions.
    // NOTE: The gap/offset in tag ID for the table data is intentional - this allows
    the insertion

```

```

        // of other operation data into the C1219 Req message proto while maintaining
backward
    // compatibility with previous operation definitions.
    optional bytes tableData = 14;

    message C1219Validation
    {
        // This flag essentially inverts the validation conditional logic check
        // of all validation statements - e.g., integerReadData != number,
        // readHexString != hexStringCompare will result in a successful validation
        // check when isNot==1. If not defined, isNot is assumed to be false.
        optional bool isNot = 1;

        // The C12.19 operation response is converted to an integer and must
        // equate to the following value to pass the validation step
        optional int64 number = 2;

        // The C12.19 operation response is converted to an integer and cannot be
        // less than the specified lower range value.
        optional int64 rangeLower = 3;

        // The C12.19 operation response is converted to an integer and cannot be
        // more than the specified upper range value.
        optional int64 rangeUpper = 4;

        // The C12.19 operation response must match the specified hex string using
        // a standard strcmp library call
        optional bytes hexStringCompare = 5;

        // An alternative C12.19 Program Validation method can be used to test
        bitfields
        // within a single table byte. However, there are a couple of restrictions
        regarding
        // how this is set up in the validation submessage:
        // - HexStringCompare array cannot be defined (it will trump any tags that
        follow it).
        // - If the associated READ operation contains multiple read bytes (FULL or
        PARTIAL),
        //   only the first bytes is tested with the specified bitFieldMask. Is is
        expected that
        //   the corresponding table read is a PARTIAL single byte read (at a
        specified offset)
        // - The hexBitFieldMask cannot span multiple byte boundaries and is limited
        //   to an 8-bit mask (i.e., any MSB above 0xFF are dropped).
        // - The bitFieldValue is always tested as an integer. Obviously, it only
        //   makes sense if it is in the range of the set by the BitMask.
        // - hexBitFieldMask and bitFieldValue must BOTH be defined (or it is treated
        //   as a validation decode failure)
        optional uint32 hexBitFieldMask = 7;
        optional uint32 bitFieldValue = 8;
    }

    // Program validation data - required for COMPATIBILITY and AUDIT action
    // types, ignored by STANDARD actions or C12.19-level accesses that do not
include
    // meter response data (i.e., a WRITE_* operation) - NOTE: Multiple validation
    // options are allowed/checked, but common sense needs to prevail when determining
    // the range checks.
    optional C1219Validation validation = 21;
}

// A chained list of C1219 Operation request to be executed in order
repeated C1219Operation operations = 20;
}

```

The following bullet items provides more details on each of the C1219Req message fields as it pertains to C12.19 Program Sequence execution in the MCM firmware:

- **corrId** – The Correlation ID that is extracted is used in the corresponding C1219 Response message construction. Unlike other AMI message, if no correlation ID, the Program Execution handlers will default to a value of zero and proceed with the operation (as opposed to generating a run-time assert).
- **startTime** – POSIX time representation of the GMT network time at which point the C12.19 Program Sequence is to be started. If this field is not defined, then the MCM firmware will execute the sequence immediately.
- **expiryTime** – POSIX time representation of the GMT network time at which point the C12.19 Program Sequence is considered invalid. The most likely occurrence of this scenario is if the MCM unit enters into a power-fail state and drops off the network while a pending C1219Req is buffered. Under these scenarios and when the unit recovers, the program sequence will not execute and the C1219Rsp will be generated with the appropriate expired result code. If this field is not defined, then the MCM firmware will buffer a pending C1219Req indefinitely (at least until aborted by the network components).
- **numRetries** – Should a C12.19 Program Sequence end with an AMI result code other than SUCCESS, the *numRetries* field will gate additional retry attempts by re-enqueuing the PROGRAM_SEQUENCE event to the METER layer. There are, however, some exceptions to the Retry rule – most notably, a failure caused by a COMPATIBILITY validation failure.

- **rspVerbosityType** – Further defined by the following:

```
// Enumeration of the response verbosity type that specifies how the MCM will
// build
// the C1219Rsp message.
enum C1219RspVerbosityType
{
    C1219_RSP_VERBOSITY_TYPE_NIL                = 0;

    // C12.19 Minimal verbosity response type - only the overall C1219 Program
    // Sequence result
    // code is returned
    C1219_RSP_VERBOSITY_TYPE_MIN                = 1;

    // C12.19 Operation Failure details response type - in addition to the overall
    // C1219 Program
    // Result, the MCM will details on each failed C12.19 Operation.
    C1219_RSP_VERBOSITY_TYPE_FAILURE_DETAILS    = 2;

    // C12.19 Maximum verbosity response type - in addition to the overall C1219
    // Program Result,
    // the MCM will provide details on all C12.19 Operations.
    C1219_RSP_VERBOSITY_TYPE_MAX                = 3;
}
```

The formation of the C1219 Operation responses will depend on the verbosity setting (see next section).

- **retryDelay** – A user specified retry delay (in seconds) between C12.19 Program Sequence retry attempts. If not specified in the C1219Req, the MCM firmware will use a default delay value of 15seconds.
- **operations** – The C12.19 Operation must have an *action* identified as follows:

```
// Enumeration of possible actions for C12.19-level accesses.
enum C1219Action
{

```

```

C1219_ACTION_NIL          = 0;

// A C12.18 table read (FULL or PARTIAL based on supplemental operation
fields).
C1219_ACTION_READ        = 1;

// A C12.18 table write (FULL or PARTIAL based on supplemental operation
fields).
C1219_ACTION_WRITE       = 2;

// A C12.19 procedure call (via ST7 table write).
C1219_ACTION_PROCEDURE    = 3;

// A C12.19 SP10 with variables as determined at other times
C1219_SET_TIME            = 4;

// A delay operation to enforce between C12.19-level accesses by EMC
C1219_DELAY               = 5;

// A C12.19 patch operation to precede a WRITE or PROCEDURE action to allow
the host
// to replace a segment of table data with run-time data from a C12.19 level
access
C1219_PATCH               = 6;
// A C12.19 table read-modify-write in which a specified bitfield (offset and
mask) is
// modified with a specified bitfield value. These READ/WRITE table actions
are always PARTIAL
// and only operate on a single specified byte (by offset) within a specified
Meter table.
// The other bitfields in the Table byte are unchanged by this operation
C1219_READ_MODIFY_BITFIELD_WRITE = 7;
}

```

In the C12.19 Program Sequence implementation, the *operation* describes the C12.19 Level access that is validated and executed by the METER layer components. The METER layer run-time executable treats the C12.19 Action as an atomic operation allowing other MCM executables a chance to run before proceeding to the next sequential action.

4.1.2 C12.19 Response Message

The C1219Rsp Message, using the C12_19_RSP AMI message ID, is defined by the following proto definition:

```

message C1219Rsp
{
    // The correlation ID from the request.
    optional uint32 corrId = 1;

    // POSIX timestamp of when the device started servicing the request.
    optional uint32 startTime = 2;

    // The response verbosity of the C1219Rsp (i.e., this message)
    optional C1219RspVerbosityType rspVerbosityType = 4;

    message C1219OperationResult
    {
        // The action that was requested.
        optional C1219Action action = 1;

        // The actionType to further refine how the program operation was processed - if
        // not defined, the default is a standard PROGRAM_AND_READ action type.
        optional C1219ActionType type = 2;

        // The patch type - only applicable to PATCH operations.
    }
}

```

```

optional C1219PatchType patchType = 3;

// The type of table to act on.
optional ami.defs.MeterTableType tableType = 4;

// The table number to act on for READ_*, WRITE_* actions.
optional uint32 number = 5;

// For READ_* and WRITE_* actions, explicitly define if it is a FULL or PARTIAL
action
optional bool isFull = 6;

// The offset to read from or write to for READ_PARTIAL, WRITE_PARTIAL, or PATCH
actions.
optional uint32 offset = 7;

// For READ_FULL action this is the maximum data to respond with.
//
// For READ_PARTIAL action this is the size of the partial read.
//
// For a PATCH action, this is the size of the patch data to replace with run-
time C12.19 data
optional uint32 maxLength = 8;

// C12.18 result code.
optional uint32 c1218ResultCode = 9;

// Result Code in Standard Table 8
optional uint32 C1219ProcedureResultCode = 10;

// Procedure Response Record in Standard Table 8
optional bytes C1219ProcedureRespRcd = 11;

// Included for READ_FULL, READ_PARTIAL;
// NOTE: The gap/offset in tag ID for the table data is intentional - this allows
the insertion
// of other operation data into the C1219 Rsp message proto while maintaining
// backward compatibility with previous result definitions.
optional bytes tableData = 20;
}

repeated C1219OperationResult results = 20;

// The overall result code.
optional ami.defs.ResultCode resultCode = 21;

// POSIX timestamp of when the device finished the request.
optional uint32 endTime = 22;

// Number of times program retried from start (0 - single pass)
optional uint32 numRetriesUsed = 23;
}

```

The following bullet items provides more details on each of the C1219Rsp message fields as it pertains to C12.19 Program Sequence execution in the MCM firmware:

- **corrId** – The Correlation ID is taken directly from the C1219Req message.
- **startTime** – POSIX time representation of the GMT network time at which point the C12.19 Program Sequence is executed as tracked by the MCM firmware.
- **results** – The result of a C1219 Level Access is reported by this submessage depending on the verbosity type:
 - Min verbosity – Only results for an explicit READ action are reported.

- Failure Details verbosity – In addition to explicit READ actions, any operation that fails will have details on its operation type.
- Max verbosity – Each C12.19 action will have its operation reported in the order it is processed with the ANSI Meter.
- **endTime** – POSIX time representation of the GMT network time at which point the C12.19 Program Sequence is finished – as tracked by the MCM firmware.
- **numRetriesUsed** – Self-explanatory – the number of times the C12.19 Program Sequence was executed on target before the OTA generation of the C12.19Rsp message.

4.1.3 C12.19 Abort Message

The C12.19Abort Message, using the C12_19_REQ_ABORT AMI message ID, is defined by the following proto definition:

```
message C1219AbortReq
{
    // The correlation ID of the C1219Req/Rsp to abort. If not specified, this
    // aborts/cancels the current request regardless of corrId.
    optional uint32 corrId = 1;
}
```

Because the C12.19 Abort message is received over the dedicated resumable USP Stream (i.e., StreamID#10), it will overwrite any queued/pending C12.19Req message in external NVM essentially “aborting” it even without having processed by the MCM Application Layer.

If there is no pending C12.19Req message in external NVM, the C12.19Abort is essentially ignored (other than a warning log statement reported over the Host Serial interface (see next section)).

4.1.4 AMI Result Definitions

The following AMI result definitions have been defined in <ami_defs.proto> and are *exclusive* to run-time exceptions encountered during on-target C12.19 Program Sequence execution – as such, they will be reported in the C12.19Rsp message:

- PROGRAM_SEQUENCE_OPERATION_DECODE_FAILURE
- PROGRAM_SEQUENCE_INVALID_OPERATION
- PROGRAM_SEQUENCE_TIME_EXPIRE
- PROGRAM_SEQUENCE_COMPATIBILITY_FAILURE
- PROGRAM_SEQUENCE_AUDIT_FAILURE
- PROGRAM_SEQUENCE_VALIDATION_DECODE_FAILURE
- PROGRAM_SEQUENCE_NUM_EXCEPTION_RETRIES_EXCEEDED
- PROGRAM_SEQUENCE_COMPLETED_WITH_OPERATION_LEVEL_FAILURE

The following AMI result definitions have been defined in <ami_defs.proto> and are *exclusive* to run-time exceptions regard C12.19 Program Sequence accesses to external Flash where the

C1219Req and C1219Rsp message buffers are located – as such, they will be reported in the MCM Error Indicators:

- C1219REQ_NVM_ACCESS_FAILURE
- C1219RSP_NVM_ACCESS_FAILURE
- C1219RSP_BUFFER_ALLOCATION_FAILURE

The following AMI result definitions have been defined in <ami_defs.proto> and are *exclusive* to run-time exceptions regard C12.19 Program Sequence interactions with the HOST_CMN USP stream protocol – as such, they will be reported in the MCM Error Indicators:

- DL_RESUMABLE_STREAM_MSG_INTEGRITY_FAILURE
- DL_RESUMABLE_STREAM_INVALID_MSG_ID
- UL_RESUMABLE_STREAM_START_FAILURE

4.2 Host Interface

The C12.19 Program Sequence software components makes extensive use of host logging features to monitor the following:

- From the *host_cmm_app.c* module:
 - The status of all downlink data fragments over Stream10 as they are received and committed to external NVM (i.e., the C1219Req message)
 - The status of all uplink data fragments over Stream10 as they are retrieved from external NVM and transmitted to the node (i.e., the C1219Rsp message).
 - The detection and follow-up processing of C1219Req overwrites – i.e., the C1219 Abort mechanism.
- From the *ami_proto_mcm.c* module:
 - The status of initial C1219 Program Sequence message handling from the MCM APP layer – this includes the rescheduling of a C1219 Program Sequence event due to a delayed start time.
 - All Decoding/Encoding errors that will likely lead to a failed Program Sequence execution.
 - The status of any C1219 Validation processing.
- From the *meter.c* module:
 - That status of each iterative pass through the meter-layer run-time executable that results in an atomic C1219-level access.
 - The status of each callback following an atomic C1219-level access.
 - All METER layer exception that occur during a C1219 Program Sequence execution.

In addition to logging, the DOM software unit provides the following monitor interfaces with the external Host PC:

NOTE: Several of the C12.19 test message require use of the python protobuf component to be installed with the standard python library.

4.2.1 C12.19 Request Download Message

Using this host↔node message configuration, an external user can verify the C1219Req message buffer in external NVM – including an option to decode and display each Protobuf field. However, this should only be used when the C1219Req data is pending via the *startTime* message field. On the EMCM platform, this is supported by using the following python script:

- *emcm_test_c1219_req_download.py*

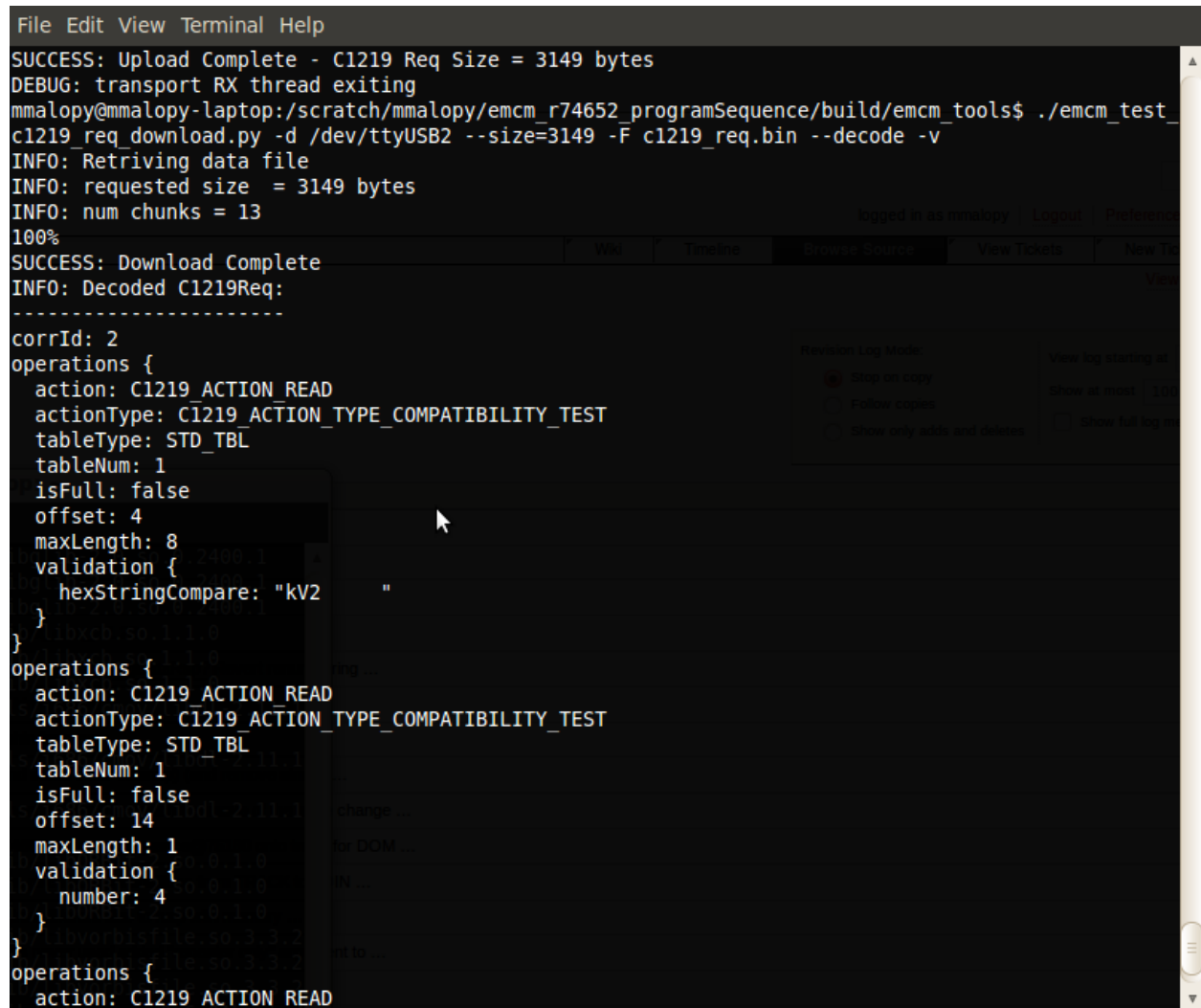
The following message-specific command line arguments are used (as reported via the `--help` command line option – these are in addition to the standard EMCM message arguments):

- *-F, --file=* - The destination filename of the extracted AMI Protobuf C1219Req message.
- *-s, --size=* - A specified size, in bytes, of the AMI Protobuf C1219Req message – this is useful for incomplete message transfers where the total size of the message is unknown by the EMCM firmware.
- *-a, --autosize* – Auto determined size, in bytes, of the AMI Protobuf C1219Req message – this will only work if the a completed C1219Req message has been received and loaded into the EMCM's external NVM.
- *-D, --decode* – An optional setting that enables and displays Protobuf decoding of the extracted C1219Req message. NOTE: The Protobuf methods for displaying hex data strings are malformed (it expects ASCII encoded characters) – string data arrays are better displayed in their raw format by analyzing the file output.

This message will provide the following information:

- Download status
- Command failure information (if it did not succeed).
- If decoding is enabled, the decoded C1219Req message.

The following screenshot shows the response output from the host application for this message request as generated by the EMCM application:



```

File Edit View Terminal Help
SUCCESS: Upload Complete - C1219 Req Size = 3149 bytes
DEBUG: transport RX thread exiting
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$ ./emcm_test_c1219_req_download.py -d /dev/ttyUSB2 --size=3149 -F c1219_req.bin --decode -v
INFO: Retriving data file
INFO: requested size = 3149 bytes
INFO: num chunks = 13
100%
SUCCESS: Download Complete
INFO: Decoded C1219Req:
-----
corrId: 2
operations {
  action: C1219_ACTION_READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 4
  maxLength: 8
  validation {
    hexStringCompare: "kv2"
  }
}
operations {
  action: C1219_ACTION_READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 14
  maxLength: 1
  validation {
    number: 4
  }
}
operations {
  action: C1219_ACTION_READ

```

Figure 4. C12.19 Request Download Example

4.2.2 C12.19 Response Download Message

Using this host↔node message configuration, an external user can verify the C1219Rsp message buffer in external NVM – including an option to decode and display each Protobuf field.

However, this should only be used when the C1219Rsp data is fully assembled. NOTE: That the C1219Rsp message will remain in external NVM even after it has been delivered to the network components over USP – however, the buffer will be cleared whenever the next C1219 Program Sequence is initiated in the EMCM’s METER Layer. On the EMCM platform, this is supported by using the following python script:

- *emcm_test_c1219_req_download.py*

The following message-specific command line arguments are used (as reported via the `--help` command line option – these are in addition to the standard EMCM message arguments):

- `-F, --file=` - The destination filename of the extracted AMI Protobuf C1219Rsp message.

- *-s, --size=* - A specified size, in bytes, of the AMI Protobuf C1219Rsp message – this is useful for malformed messages during encoding which the EMCM will deliver an error indicator over the non-resumable uplink channel for.
- *-a, --autosize* – Auto determined size, in bytes, of the AMI Protobuf C1219Rsp message – this will only work if the a completed C1219Rsp message is loaded into the EMCM's external NVM.
- *-D, --decode* – An optional setting that enables and displays Protobuf decoding of the extracted C1219Rsp message. NOTE: The Protobuf methods for displaying hex data strings are malformed (it expects ASCII encoded characters) – string data arrays are better displayed in their raw format by analyzing the file output.

This message will provide the following information:

- Download status
- Command failure information (if it did not succeed).
- If decoding is enabled, the decoded C1219Rsp message.

The following screenshot shows a partial response output from the host application for this message request as generated by the EMCM application (the response can be quite long):

```

File Edit View Terminal Help
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$ ./emcm_test_c1219_rsp_download.py -d /dev/ttyUSB2 --autosize -F c1219_rsp.bin --decode -v
INFO: Retriving data file
INFO: requested size = 1149 bytes
INFO: num chunks = 5
100%
SUCCESS: Download Complete
INFO: Decoded C1219Rsp:
-----
corrId: 2
startTime: 1410197258
results {
  action: C1219 ACTION READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 4
  maxLength: 8
  tableData: "kV2"
}
results {
  action: C1219 ACTION READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 14
  maxLength: 1
  tableData: "\004"
}
results {
  action: C1219 ACTION READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL

```

Figure 5. C12.19 Response Download Example

4.2.3 C12.19 Request Upload Message

Using this host↔node message configuration, an external user can load a formatted C1219Req message in external NVM – including an option to decode and display each Protobuf field. This should only be used for unit testing purposes for internal development and/or testing purposes. The DOM manager supports a test option to simulate the USP complete event which will generate MCM application processing for the uploaded message (see reference [3]). On the EMCM platform, this upload command is supported by using the following python script:

- *emcm_test_c1219_req_upload.py*

The following message-specific command line arguments are used (as reported via the `--help` command line option – these are in addition to the standard EMCM message arguments):

- *-F, --file=* - The source filename of an encoded AMI Protobuf C1219Req message (see next section for a provided utility that generates this message format). NOTE: The

encoded message must also include the AMI message header with a properly calculated message CRC.

This message will provide the following information:

- Upload status
- Command failure information (if it did not succeed).
- If decoding is enabled, the decoded C1219Req message.

The following screenshot shows the response output from the host application for this message request as generated by the EMCM application:

```

File Edit View Terminal Help
maxLength: 9
tableData: "\020\030\021\001\n\004\000-\000"
}
results {
  action: C1219 ACTION READ
  actionType: C1219 ACTION_TYPE_AUDIT_TEST
  tableType: STD_TBL
  tableNum: 54
  isFull: false
  offset: 0
  maxLength: 45
  tableData: "\216\010\003B\t\0233\010\001e\t\023)\t\023:\t\023+\010\002[\014\023\341\010\023\347
\023\354\310\023\0018\001\002\230\001\002\000\021\002\0001"
}
results {
  action: C1219 ACTION READ
  actionType: C1219 ACTION_TYPE_AUDIT_TEST
  tableType: MFG_TBL
  tableNum: 67
  isFull: false
  offset: 154
  maxLength: 2
  tableData: "$w"
}
resultCode: SUCCESS
endTime: 1410197270
-----
DEBUG: transport RX thread exiting
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$ ./emcm_test_
c1219_req_upload.py -d /dev/ttyUSB2 -F kv2c_test.bin -v
INFO: Sending data file
INFO: file size = 3149 bytes
INFO: num chunks = 13
100%
SUCCESS: Upload Complete - C1219 Req Size = 3149 bytes
DEBUG: transport RX thread exiting
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$

```

Figure 6. C12.19 Request Upload Example

4.2.4 C12.19 Request Generation Utility

There is a python support utility that builds an AMI Protobuf formatted C1219Req message that can be uploaded on target through use of the C1219 Request Upload message (see previous

section). This provides local test capability on an EMCM unit that bypasses the OTA C1219 Request transfer over the ORW wireless network. It is primarily used for development unit testing purposes. On the EMCM platform, this upload command is supported by using the following python script:

- *emcm_test_c1219_req_gen.py*

The following script-specific command line arguments are used (as reported via the `--help` command line option – these are in addition to the standard EMCM message arguments):

- `-C, --corrID` – The user-specified correlation ID value that will be encoded into the C1219Req message (see section 4.1.1).
- `-s, --startTime=` – The user-specified start time value that will be encoded into the C1219Req message (see section 4.1.1). NOTE: This 32-bit value is a POSIX representation of the GMT network time at which point the EMCM will execute the C12.19 Program Sequence.
- `-e, --expiryTime=` – The user-specified expiration time value that will be encoded into the C1219Req message (see section 4.1.1). NOTE: This 32-bit value is a POSIX representation of the GMT network time at which point the EMCM will invalidate the C12.19 Program Sequence.
- `-R, --numRetries=` – The user-specified number of retries value that will be encoded into the C1219Req message (see section 4.1.1).
- `-X, --xml=` – The source .xml filename that contains the xml elements that describe an C12.19 Program Sequence that is compatible with ORW script format processing (e.g., GEC1219ScriptFormat scheme). The *test/C1219* subdirectory in the EMCM build contains a number of these .xml files used during development.
- `-A, --amiHeader` – This option prepends an AMI header (with a calculated CRC32) to the encoded C1219Req message.
- `-o, --output=` – The destination filename of the encoded AMI Protobuf C1219Rsp message.

This message will provide the following information:

- Generation failure information (if it did not succeed).
- The encoded C1219Req message.

The following screenshot shows the response output for this utility function:

```

File Edit View Terminal Help
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$
mmalopy@mmalopy-laptop:/scratch/mmalopy/emcm_r74652_programSequence/build/emcm_tools$ ./emcm_test
c1219_req_gen.py --corrId=2 -A --xml=../../test/C1219_schema/kV2c_test.xml --output=kV2c_test.bin
-v
INFO: Decoded C1219 Req fields:
corrId: 2
operations {
  action: C1219_ACTION_READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 4
  maxLength: 8
  validation {
    hexStringCompare: "KV2
  }
}
operations {
  action: C1219_ACTION_READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 14
  maxLength: 1
  validation {
    number: 4
  }
}
operations {
  action: C1219_ACTION_READ
  actionType: C1219_ACTION_TYPE_COMPATIBILITY_TEST
  tableType: STD_TBL
  tableNum: 1
  isFull: false
  offset: 15

```

Figure 7. C12.19 Request Generation Example

5 C12.19 Program Sequence Internal Interfaces

5.1 HOST_CMN Layer APIs

The following APIs are used to support the handling of USP C1219Req/C1219Rsp message fragments over resumable Stream10:

5.1.1 HOST_CMN_APP_USP_RxMsgFrag()

This is the handler for all Downlink fragments received over USP – both the traditional non-resumable message fragments as well as the persistent resumable message fragments used to support the OTA messages associated with the C12.19 Program Sequence implementation.

Regardless of the stream id/type, this API will take the following parameter(s):

- A reliable Boolean flag (C1219Req fragments will have this set to true).
- A resumable Boolean flag (C1219Req fragments will have this set to true as well).
- A stream ID (C1219Req fragments are received over Stream ID#10).
- A fragment offset – the byte offset into the C1219Req message where the current fragment starts at.
- A fragment pointer – the memory location where the current fragment resides as managed by the HOST_CMN USP component.
- The length of the fragment (in bytes).
- The total length of the entire assembled message – only set when the last fragment is received (otherwise it will be set to zero).
- A complete Boolean flag – this is set to True when all fragments for the current message have been delivered by USP.

This API will return the following:

- The Boolean status of the application processing of the fragment – i.e., true when the message fragment has been successfully copied into application memory.

If the fragment is identified to be over the stream allocated for the C12.19 Program Sequence, it will leverage the Data Object Manager (DOM) fragment commit method to store the data into external SPI Flash at the specified offset and length.

If the last fragment of a C1219Req is indicated by the *complete* flag, then the handler will also use DOM to update the total length of the C1219Req message as part of the meta-data header in external NVM. It will also invoke the AMI method that performs initial integrity and validation checks over the message body using the AMI header.

If the fragment is overwriting a *pending* C1219Req in the MCM firmware, abort handlers will ensure the current request is flushed and the corresponding C1219 abort response is sent back to the network size components.

5.1.2 HOST_CMN_APP_USP_TxMsgFrag()

This is the handler for all Uplink fragments transmitted over USP once a stream transfer has been initiated for the C1219Rsp message – unlike the downlink path, this message of transferring uplink AMI data is unique to the C12.19 Program Sequence.

This API, when invoked by USP, will take the following parameter(s):

- A user-space defined memory handle queued during the start of the resumable transfer – However, this is not used during the C1219Rsp message transfer as the memory buffer used for the response data is in external SPI Flash and the Stream ID is sufficient enough to identify which data object to use.
- A resumable Boolean flag (C1219Rsp fragments will have this set to true).
- A stream ID (C1219Rsp fragments are transmitted over Stream ID#10).
- A fragment offset – the byte offset into the C1219Rsp message where the current fragment starts at.
- A fragment pointer – the memory location where the current fragment to be updated resides as managed by the HOST_CMN USP component.
- The length of the fragment (in bytes).

If the fragment is identified to be over the stream allocated for the C12.19 Program Sequence, it will leverage the Data Object Manager (DOM) fragment get method to retrieve the requested data from external SPI Flash and update the data in the USP RAM buffer at the specified offset and length. HOST_CMN will then forward the message data to the ORW Node through the generation of a TxSDU request over the NODE SPI interface.

5.1.3 HOST_CMN_APP_USP_ResetRxStream()

The backend components can cancel an in-progress C1219Req transfer over USP which will eventually invoke this API method on the EMCM.

If the stream is identified for use by the C12.19 Program Sequence (i.e., stream ID#10), then it will use DOM to flush/re-allocate the C1219Req buffer in external NVM for any subsequent downlink message transfers.

5.2 MCM APPLICATION Layer APIs

There are no new APIs created in the MCM_APP layer to support the C12.19 Program Sequence implementation.

However, the existing MCM application message handlers have been modified to handle both the receive C1219Req and C1219Abort messages and to invoke the corresponding AMI_PROTO handlers for performing initial decoding and, if necessary, the scheduling of the C1219Req.

5.3 AMI_PROTO C1219 Layer APIs

This is the layer that contains the bulk of the C12.19 Program Sequence software changes. The following APIs are essential to the successful decoding and encoding of both the C1219Req and C1219Rsp. Tracking variables in the AMI_PROTO_C1219 control block also ensure that message parsing and offset management are properly tracked to ensure both messages are properly formatted using the Google Protobuf methods (see reference [5]).

5.3.1 AMI_PROTO_C1219_Init()

This API is called as the MCM firmware comes out of reset and will initialize run-time critical variables within the AMI_PROTO_C1219 control block.

This API will take the following parameter(s):

- A Rx Data Handler function callback to invoke upon reception of a valid message over the Resumable Stream Handler (see next section).

5.3.2 AMI_PROTO_C1219_RxResumableStreamHandler()

This API is provided as the first validation step of a completed message in the C1219Req buffer in external NVM. It is responsible for performing a CRC calculation over the complete C1219Req message and comparing the results with the AMI Header CRC. If the CRC check passes, the message is then enqueued to the MCM_APP layer for further processing

This API will take the following parameter(s):

- The total length of the message in the C1219Req buffer.

The CRC calculation is done in a temporary 256-byte array located in stack memory using the Data Object Manager to retrieve the data fragments.

If the CRC check fails, an error indicator will be logged and returned to the back-end components over the non-resumable stream. In addition, the C1219Req buffer will be flushed and re-allocated for any subsequent DL stream fragment handling.

5.3.3 AMI_PROTO_C1219_HandleMcmAppMsgC1219Req()

This API is used to support initial decoding and scheduling of the C1219Req in the MCM_APP run-time executable..

This API will take the following parameter(s):

- A receive message descriptor – as defined by the *AmiProtoRxMsg_t* data structure – however, only the length element is used.

The following processing steps describe the action(s) taken by this API:

- Up to the first 256-byte chunk is retrieved from external NVM and copied into the AMI_PROTO's DL Decode Buffer. This is more than enough data bytes to decode all C1219Req header tags.
- A nanopb stream object is created using the data in the DL Decode Buffer.

- Using nanopb manual decode methods, each tag is manually decoded and placed into static storage until the start of the C1219 Operation array is reached.
- If the Posix startTime tag is extracted and greater than the current network time, a callback is scheduled to requeue the C1219Req event to the MCM_APP “Now” queue.
- If the Posix expiryTime tag is extracted and is equal to or less than the current network time, then the Program Sequence is aborted and a C1219Rsp is formed with the TIME_EXPIRE result code.
- If the C1219Req needs to be handled immediately, then the PROGRAM_SEQUENCE event is enqueued to the METER layer.

5.3.4 AMI_PROTO_C1219_DecodeOperation()

This API is used by the METER layer to decode each a single C1219 Operation from the C1219Req buffer in external NVM. Tracking variables in the AMI_PROTO control block determine at which offset into the buffer to begin decoding.

This API will take the following parameter(s):

- A pointer to a C12.19 Operation Descriptor, as defined by the *AMI_PROTO_C1219OperationDescriptor_t* struct – this API will update elements of this descriptor as they are decoded for the current parsed operation.

This API will return the following:

- The Boolean status of the C1219 decoding operation – i.e., true when the operation decoded successfully - false if an error occurred during decoding.

The following processing steps describe the action(s) taken by this API:

- Up to the first 256-byte chunk is retrieved from external NVM and copied into the AMI_PROTO’s DL Decode Buffer.
- A nanopb stream object is created using the data in the DL Decode Buffer.
- Using nanopb manual decode methods, each operation tag is manually decoded into the C1219 Operation Descriptor location as provided by the caller. Decoding will halt at the start of either the Table Data array or Validation Data submessage and tracking variables will be updated accordingly to support subsequent decoding of these two data array types.
- The AMI_PROTO tracking variables will update the offset variables into the C1219Req Buffer for each C1219 Operation decoded.

5.3.5 AMI_PROTO_C1219_DecodeTableData()

This API is used by the METER or C1218_LINK Layer when it is necessary to extract table data from the C1219Req buffer in external NVM for the purposes of TABLE or PROCEDURE table writes to the ANSI meter.

This API will take the following parameter(s):

- A data pointer in RAM to extract the requested table data to.
- An offset into the C1219 operation table data array to begin the transfer from.
- A requested length segment of the C1219 operation table data.

This API will return the following:

- The number of table data bytes that have been transferred starting at the specified data pointer in RAM.

The following processing steps describe the action(s) taken by this API:

- Range check the AMI_PROTO tracking variables to ensure the C1219Req offset tracking variables have decoded the table data tag and there are remaining table data bytes extract, if no table data can be decoded, a zero length is returned.
- With the aid of the Data Object Manager meta-data and AMI_PROTO tracking variables, the requested data will be copied from external Flash to the designated location in RAM specified by the caller.
- If the AMI_PROTO has decoded and prepared patch data for the current C1219 Operation, then the corresponding patch segment will be extracted from the Bulk Read buffer in external NVM also through the use of DOM – the patch data will overwrite the extracted table data bytes.
- The AMI_PROTO tracking variables will update the offset variables into the C1219Req Buffer for each extracted table data bytes.

5.3.6 AMI_PROTO_C1219_FlushTableData()

This API is used to advance the AMI_PROTO table data tracking variables past any unprocessed Table Data bytes in the C1219Req buffer from external NVM. This is used only during decoding errors of WRITE or PROCEDURE data in the METER layer – as a result, it is normally not called during a C12.19 Procedure Sequence.

This API will return the number of table data bytes that have been flushed in this manner.

5.3.7 AMI_PROTO_C1219_GetRemainingC1219ReqSize()

This API is used as a utility function for the METER layer to determine the number of remaining bytes left to decode in the C1219Req buffer in external NVM as tracked by the AMI_PROTO control block variables. When the C12.19 Program Sequence is successfully decoded and executed each C12.19 Level-Access, then there will be zero bytes left in the message buffer.

The API will return the number of C1219Req bytes to decode based on these tracking variables.

5.3.8 AMI_PROTO_C1219_GetRemainingTableDataSize()

Similar to the previous API, the utility function returns the number of unprocessed Table Data bytes for the *current* C12.19 Operation. It is used to support C12.18 packet data formation as well as exception/failsafe handling following the execution of any WRITE or PROCEDURE action.

5.3.9 AMI_PROTO_C1219_AllocateResponse()

This API is used by the METER Layer when it initially acts upon the enqueued PROGRAM_SEQUENCE event. It will allocate a “fresh” C1219Rsp buffer in external NVM for the formation of the C1219 Response message and encode both the start time and initial response parameters.

This API will take the following parameter(s):

- The correlation ID, as extracted from the C1219 Request message.

This API will return the following:

- The Boolean status of the C1219 Response allocation operation – i.e., true when the buffer was allocated successfully - false if an error occurred during NVM allocation.

The following processing steps describe the action(s) taken by this API:

- Using the Data Object Manager, a new “slot” for the C1219Rsp buffer is allocated. If necessary, the DOM will perform a bulk erase data over this new “slot” to prepare the buffer for subsequent write updates.
- Using nanopb methods, the C1219 correlation ID is encoded and then committed to external NVM using DOM. AMI_PROTO tracking variables will keep track of the current offset into the C1219Rsp following each encoded tag object.
- Using nanopb methods, any remaining C1219Rsp data which precede the C1219 Operation response are also encoded in the same manner as the C1219 correlation ID.

5.3.10 AMI_PROTO_C1219_AllocateReadStorage()

Also as an initial step in the METER Layer’s processing of the enqueued PROGRAM_SEQUENCE event, a scratch area in external NVM will be allocated for any read data storage as a result of a C1219 READ or PATCH action. NOTE: Bulk read cannot be saved directly into the C1219Rsp buffer due to the requirement that the size of the read table array be known up-front when creating Protobuf submessages.

This method requires no parameters and simply uses the Data Object Manager to allocate an erased “slot” in external NVM for the storage of said data.

The Boolean status of the Bulk Read allocation operation is returned.

5.3.11 AMI_PROTO_C1219_PrepOperationRsp()

This API is used by the METER Layer during C1219 Operation initial processing where it forms a C1219 Operation response based on the current operational parameters. This initial response data is encoded into a nanopb output stream in RAM where it is staged until the actual execution and/or meter read data can be processed as a result of the C1219 level access.

This API will take the following parameter(s):

- A pointer to a C12.19 Operation Descriptor, as defined by the *AMI_PROTO_C1219OperationDescriptor_t* struct – This structure will have already been

updated by a call to the C1219 Operation Decode API (see section 5.3.4) that describes the current C1219 Operation being executed.

This API will return the following:

- The Boolean status of the C1219 prep operation – i.e., true when the operation encoded successfully - false if an error occurred during encoding.

The following processing steps describe the action(s) taken by this API:

- The C1219Rsp prep buffer located in AMI_PROTO's control block is cleared and initialized as a nanopb output stream object.
- Using nanopb methods, each populated tag from the C1219 Operation Descriptor is manually encoded into the nanopb output stream. The intrinsic tracking variables provided by this stream object will keep tracking of the number of encoded bytes.

NOTE: The encoded bytes are *NOT* saved to external NVM at this time. It is temporary staged into RAM until the results of the C1219 Operation are known.

5.3.12 AMI_PROTO_C1219_CommitBulkReadData()

This API is provided as a utility function in support of streaming C1219 read table data from the C1218_LINK Layer to transfer byte data from each received C12.18 packet to the Bulk Read buffer in external NVM. It also supports the temporary storage of run-time determined C12.19 PATCH data used to replace a section of WRITE or PROCEDURE table data.

This API will take the following parameter(s):

- A pointer in RAM where read data is located (i.e., the C12_18 Layer packet buffer).
- The number of read data bytes to copy beginning at the RAM byte location.

This API will return the following:

- The Boolean status of the bulk read data commit operation – i.e., true when the data is committed successfully - false if an error occurs during this process.

The following processing steps describe the action(s) taken by this API:

- Using the Data Object Manager, the read data is copied from the specified RAM location and programmed into the Bulk Read buffer in external NVM. AMI_PROTO tracking variables will keep track of the offset into this buffer where the data will be committed to.
- Tracking variables maintained by the AMI_PROTO control block will be updated for number of read data bytes committed in this manner.

5.3.13 AMI_PROTO_C1219_FinishOperationRsp()

This API is used by the METER layer following the execution of any C12.19 Level access. If necessary, it will manage any validation data associated with a COMPATIBILITY or AUDIT subaction (usually as a result of a READ action).

This API will take the following parameter(s):

- The AMI result code of the current C12.19 Level Access (i.e., SUCCESS or a FAILURE indication).

This API will return the following:

- The AMI result code of the post-processed C12.19 Level Access – this may have changed as a result of any validation steps.

The following processing steps describe the action(s) taken by this API:

- Unprocessed patch data, as tracked by AMI_PROTO, is considered invalid and is flushed – this is not considered normal and is done as a failsafe exception handler.
- If processing an AUDIT or COMPATIBILITY subaction, any READ table data in the bulk read area of external NVM is compared with validation data in the C1219Rsp buffer and are managed by AMI_PROTO tracking variables and the use of the Data Object Manager retrieve the data from external NVM. Validation processing are perform for the following data types:
 - Validation Integer compares (e.g., firmware version integers)
 - Validation Integer ranges (e.g., firmware version ranges)
 - Validation string compares (e.g., meter type)
 - Validation bitmask tests (e.g., bit set or bit clear maks)

If necessary, the result code of validation operation will change depending on the outcome of the validation checks. NOTE: Multiple validation checks can be defined per the same READ operation.

- Now that the total size of the operation response can be determined, the C1219Rsp Prep Buffer initially setup by the C1219 Prep API (see section 5.3.11) can now be programmed into the C1219Rsp buffer in external NVM using the DOM method. AMI_PROTO control block parameters are updated based on the size of the prep data that is committed.
- If there is any accumulated read data in the Bulk Read data in external NVM, this is now copied over into the C1219Rsp buffer in external NVM also using DOM methods. The read data array is the last C1219 Operation tag element to be encoded into the C1219 Operation response submessage. As with any commits of data to the C1219Rsp buffer, the AMI_PROTO tracking variables are updated with the new message offset.

5.3.14 mcmAppMsgCallbackC1219Req()

This is the final utility function in AMI_PROTO for processing a C12.19 Program Sequence at the METER Layer. If there are no failure retries specified for the current program sequence, it encodes the final response tag elements in the C1219Rsp, releases/re-allocates the C1219Rsp buffer, and kicks of the transfer of the C12.19 Response message to the network components.

This API will take the following parameter(s):

- A context (void) pointer (not used by the C1219 Callback).

- The overall AMI result code of the Program Sequence (i.e., SUCCESS or FAILURE indicator).

There are no return values for this method.

The following processing steps describe the action(s) taken by this API:

- If the operation is *not* a success and depending on the AMI_PROTO retry tracking variables, the C1219 PROGRAM_SEQUENCE event may be enqueued once again to the METER layer.
- Final C1219Rsp data (e.g., stopTime, retry count, overall result code) are encoded via nanopb methods and committed to the C1219Rsp buffer in external flash using DOM.
- The overall total length of the C1219Rsp is now encoded as part of the C1219Rsp buffer meta-data (to aid in test extraction and management by the USP methods).
- The AMI header is prepared and encoded into the C1219Rsp buffer (at offset zero). This includes a CRC calculation over the entire C1219Rsp payload.
- A resumable transfer over Uplink Stream 10 is then scheduled with the HOST_CMN USP handlers – this will begin the transfer of C1219Rsp fragments OTA to the backend components.
- Finally, a fresh “slot” for the C1219Req buffer will be requested by the Data Object Manager to prepare for the next DL session.

5.3.15 AMI_PROTO_C1219_AbortMcmAppMsgC1219req()

This is a utility API used to support a network-specified abort process of any pending C12.19 Program Sequence event on the MCM. There are no parameters or return values associated with this method which performs the following processing step(s):

- Cancel any pending Timer callback for the C12.19 Program Sequence event that may have been setup by the initial MCM_APP C1219 Request API (see section 5.3.3).
- Flush/clear a fresh “slot” for the C1219Req buffer using the Data Object Manager.

5.3.16 AMI_PROTO_C1219_SendErrorInd()

This is a utility API used to generate a “backdoor” error indicators over the non-resumable UL stream when NVM resources are compromised as part of the normal C1219 Req/Rsp message parsing (e.g., NVM PAL access errors).

This API will take the following parameter(s):

- An AMI result code to report in the error indicator (as defined by the AMI result definitions in <ami_def.proto>).
- Supplemental 32-bit error data/information.

There are no return values associated with this method which performs the following processing step(s):

- Enqueue an error indicator request with the MCM_APP layer using the parameter data and AMI_PROTO tracking variable for the current C1219Req correlation ID.

5.4 METER Layer APIs

METER Layer support for the C12.19 Program Sequence essentially leverages the existing C12.19 Level accesses that support all other MCM application interfaces with the ANSI Meter. As a result, external interface enhancements to this software component are minimal – only two APIs are needed to manage the C12.19 PROGRAM_SEQUENCE event request from the upper layers of the MCM firmware.

5.4.1 METER_ProgramSequenceReq()

This API is used the application layer to request a C12.19 Program Sequence in the METER layer.

This API will take the following parameter(s):

- A callback to execute when the C12.19 Program Sequence is complete – this will be the AMI_PROTO C1219Req callback (see section 5.3.14).
- A context pointer containing preliminary C12.19 Program Sequence parameters.

The following processing steps describe the action(s) taken by this API:

- Allocate a METER Layer event queue slot – if the queue is full, the C12.19 Program Sequence event is lost.
- Populate the allocated METER Layer event slot with the PROGRAM_SEQUENCE event.
- Pause USP Rx Stream processing – this is a point of no return for C12.19 Program Sequence processing. If the USP DL stream handlers overwrite the C1219Req buffer with additional data fragments, C1219 Operation parsing will be corrupted leading to unpredictable results – in most cases, a malformed C1219Req error is generated and the operation is aborted.
- A run flag is sent to the METER layer so that the event queue can be serviced in its run-time executable.

5.4.2 METER_CheckProgramSequenceState()

This API is used the application layer to check to see if a C12.19 Program Sequence is pending or currently in process in the METER layer.

There are no parameter(s) associated with this API and will return the following:

- A Boolean reporting the status of the active/pending C12.19 Program Sequence.

5.5 C12.18 APP Layer APIs

There are no APIs created in the C12.18 APP and LINK layer specifically to support the C12.19 Program Execution. However, because of the need to transfer bulk data for both READ and

WRITE operations that may exceed the currently allocated RAM buffers for data transfers, callback support to AMI_PROTO to transfer data in/out of external NVM was created in the current API calls to initiate a C12.18 session.

NOTE: The details of the C12.18 APP and LINK layer implementation is beyond the scope of this document – see reference [2] for further details on this software component (if it existed).

The modified API(s) for packet data callback support include the following:

5.5.1 C12_18_APP_ReadFullTable()

An additional *C12_18_RxPacketDataCallback_t* function callback parameter has been included for callback support for read data updates.

If set to NULL, the traditional copy to a static buffer followed by an app-layer memory copy is used by the C12_18 APP/LINK layer for table read operations.

5.5.2 C12_18_APP_ReadPartialTable()

Identical change the Full Read table, an *C12_18_RxPacketDataCallback_t* function callback supports partial read data transfers via a user-defined callback.

5.5.3 C12_18_APP_WriteFullTable()

Similar the read table operations, an additional *C12_18_TxPacketDataCallback_t* function callback has been included to support write table data updates as each C12.18 packet is formed. Additional logic in the C12_18_APP Layer also has been included to execute incremental CRC calculations on a packet-by-packet basis as the data is transferred from the user-layer down to the C12_18_LINK Layer.

If set to NULL, the traditional up-front copy to static buffer and pre-calculated CRC is used for during C12_18 Write data transfers.

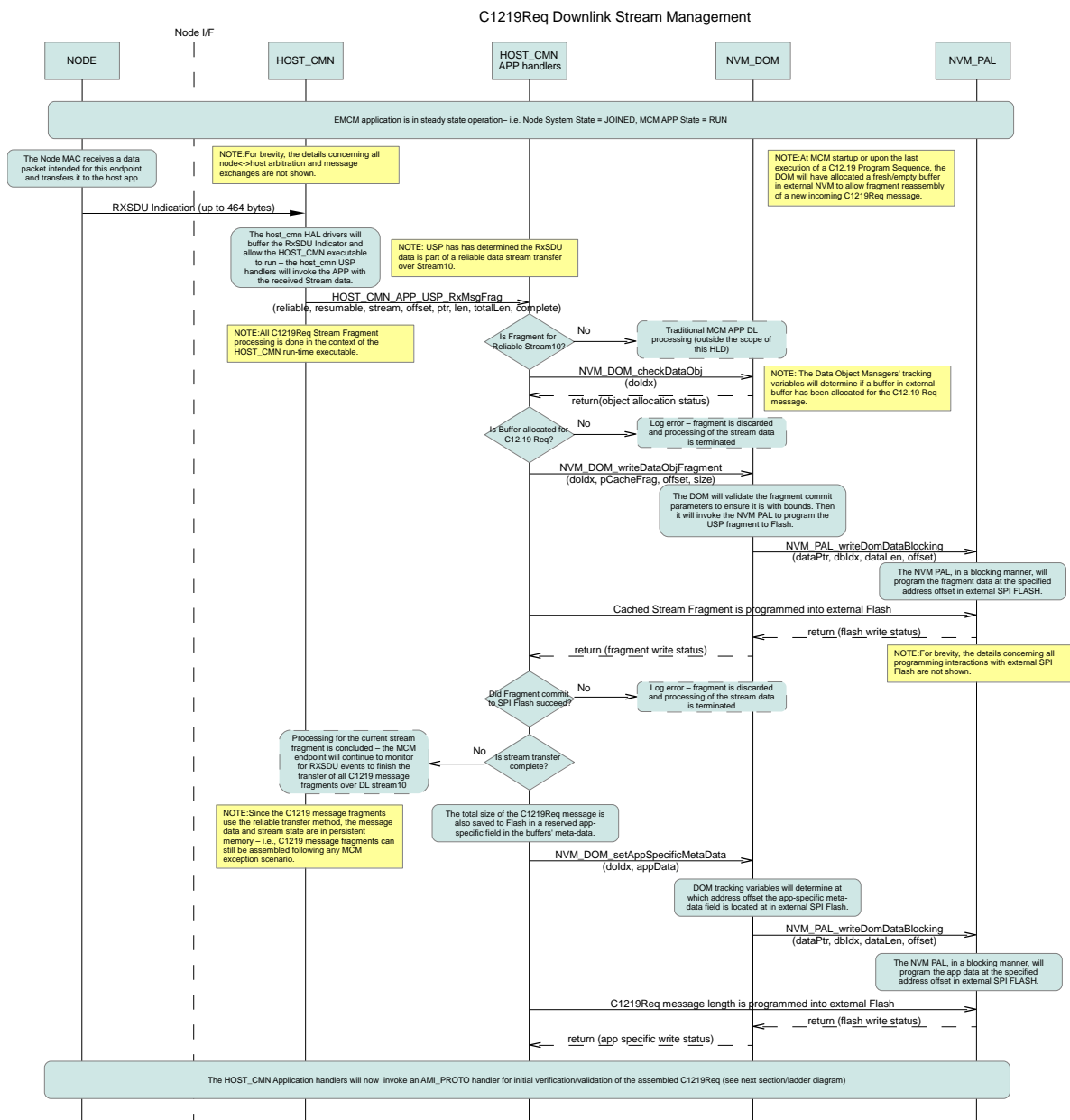
Appendix A Abbreviations and Terms

Abbreviation/Term	Definition
AP	Access Point. The ULP network component geographically deployed over a territory.
DB	Data Block
DO	Data Object
DOM	Data Object Manager
EMS	Element Management System. The network component that provides a concise view of the ULP network for controls and alarms.
FMC	Flash Memory Controller. A Kinetis module responsible for managing Program Flash, Data Flash, and/or FlexNVM.
GW	Gateway. The network appliance that provides a single entry point into the back office for the ULP network. A gateway talks upstream to the EMS and CIMA. It talks downstream to multiple APs.
HAL	Hardware Abstraction Layer. A software functional block on the ACM used to implement drivers for all ULP node interfaces.
HLD	High Level Design
IRQ	Interrupt Request
MCU	Microcontroller Unit
Node	The generic term used interchangeably with an end point On-Ramp Wireless device.
NVM	Non-Volatile Memory. In the context of the Kinetis, memory that is persistent across the different Low Leakage power modes.
ORW	On-Ramp Wireless
OTA	Over-the-Air
RAM	Random Access Memory
RTOS	Real Time Operating System
ULP	Ultra-Link Processing™. The On-Ramp Wireless' proprietary wireless communication technology.
UNIL	ULP Node Interface Library. The On-Ramp Wireless software library containing all of the necessary software interfaces to communicate with the On-Ramp Wireless Node.
uNode	Shorthand for microNode. The microNode is a second generation, small form factor, ULP wireless network module developed by On-Ramp Wireless that works in combination with various devices and sensors and communicates data to an Access Point.
UTC	Universal Time Coordinated. The time standard maintained on the On-Ramp wireless network.

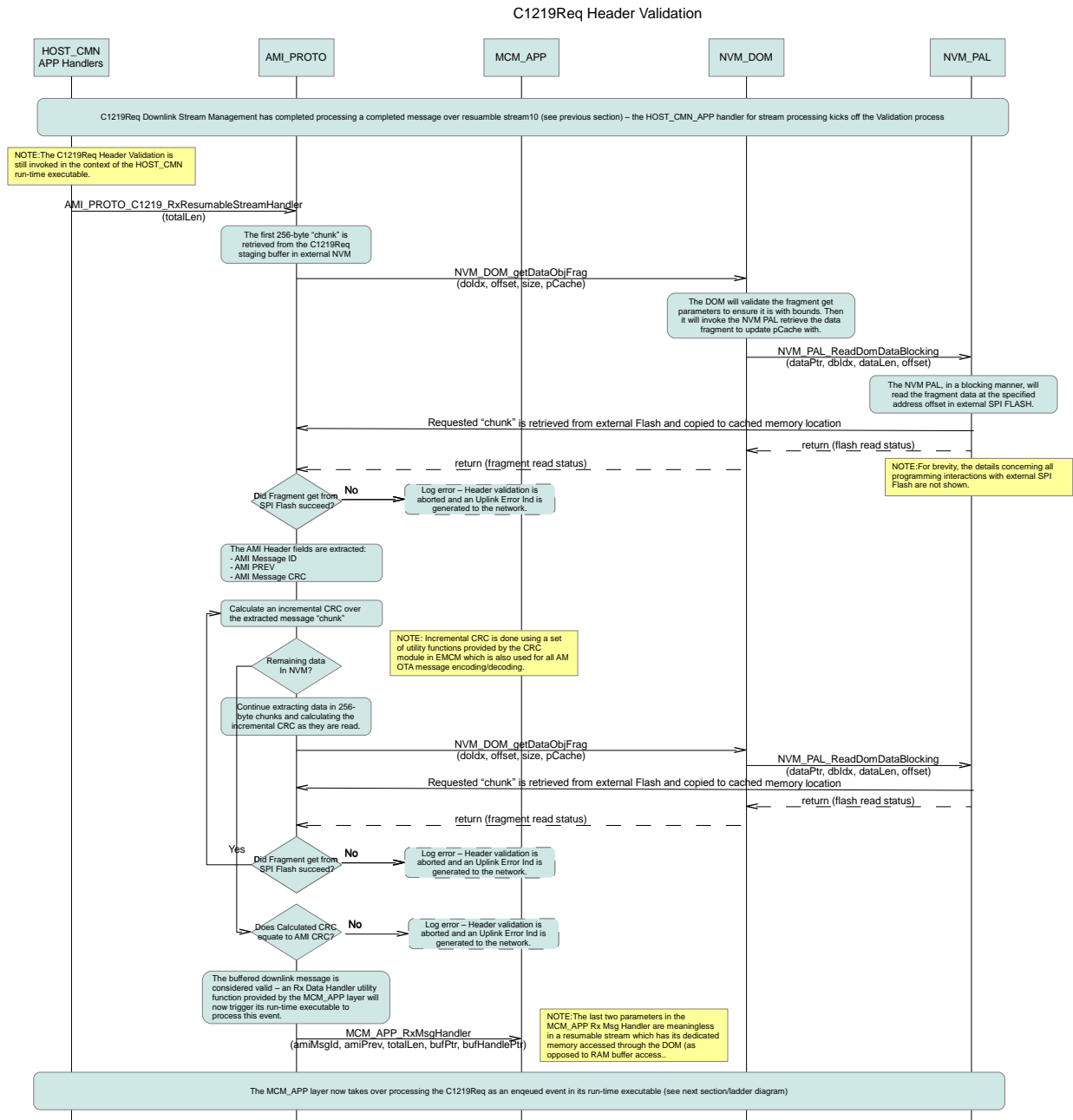
Appendix B C12.19 Program Sequence Flow Charts

This appendix can be referenced to illustrate the interactions between the various EMCN run-time executables and its supporting utility functions. These flow charts/ladder diagrams can also be view natively in .odg format as directed to by reference [8].

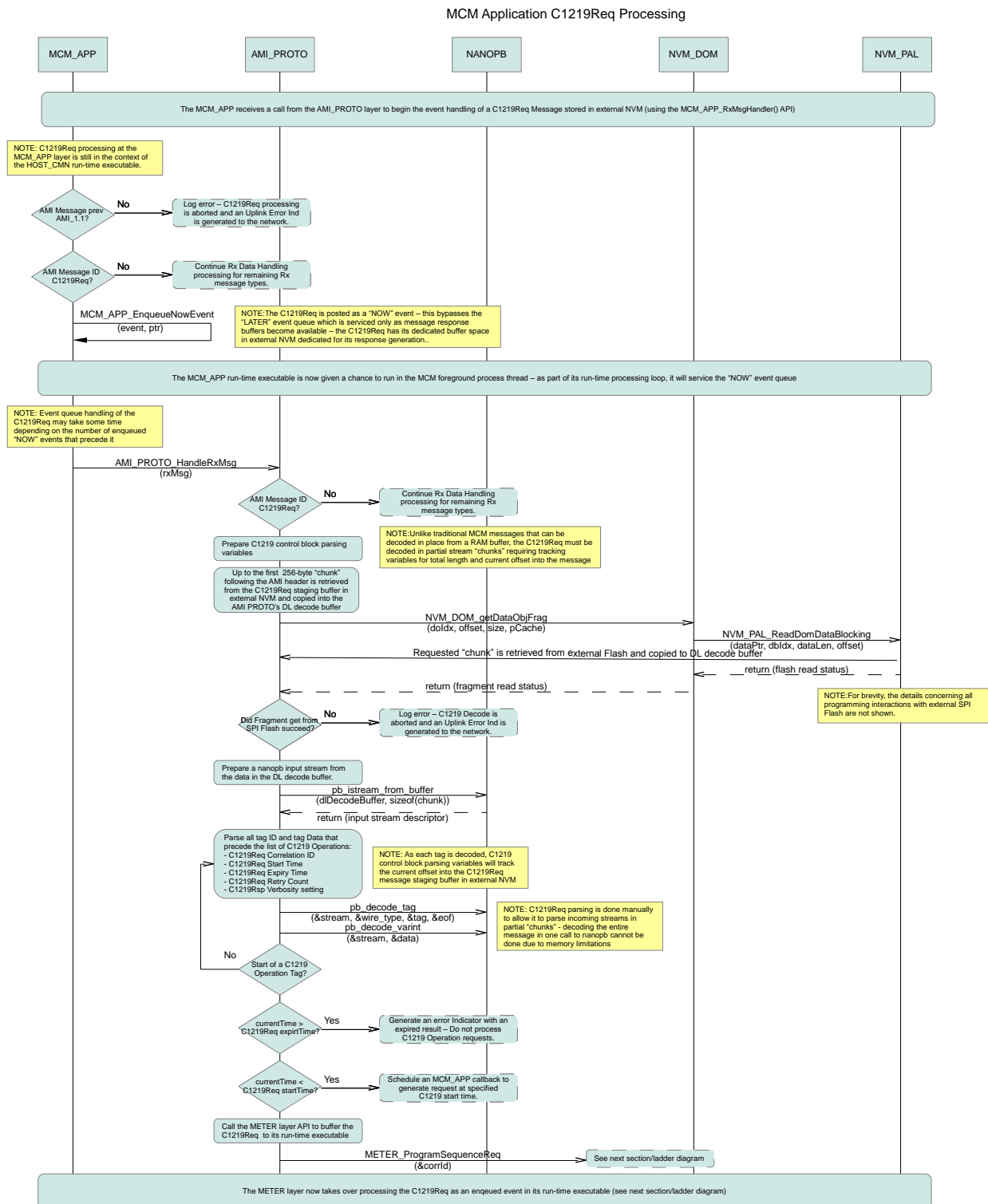
B.1 C1219Req Downlink Stream Management



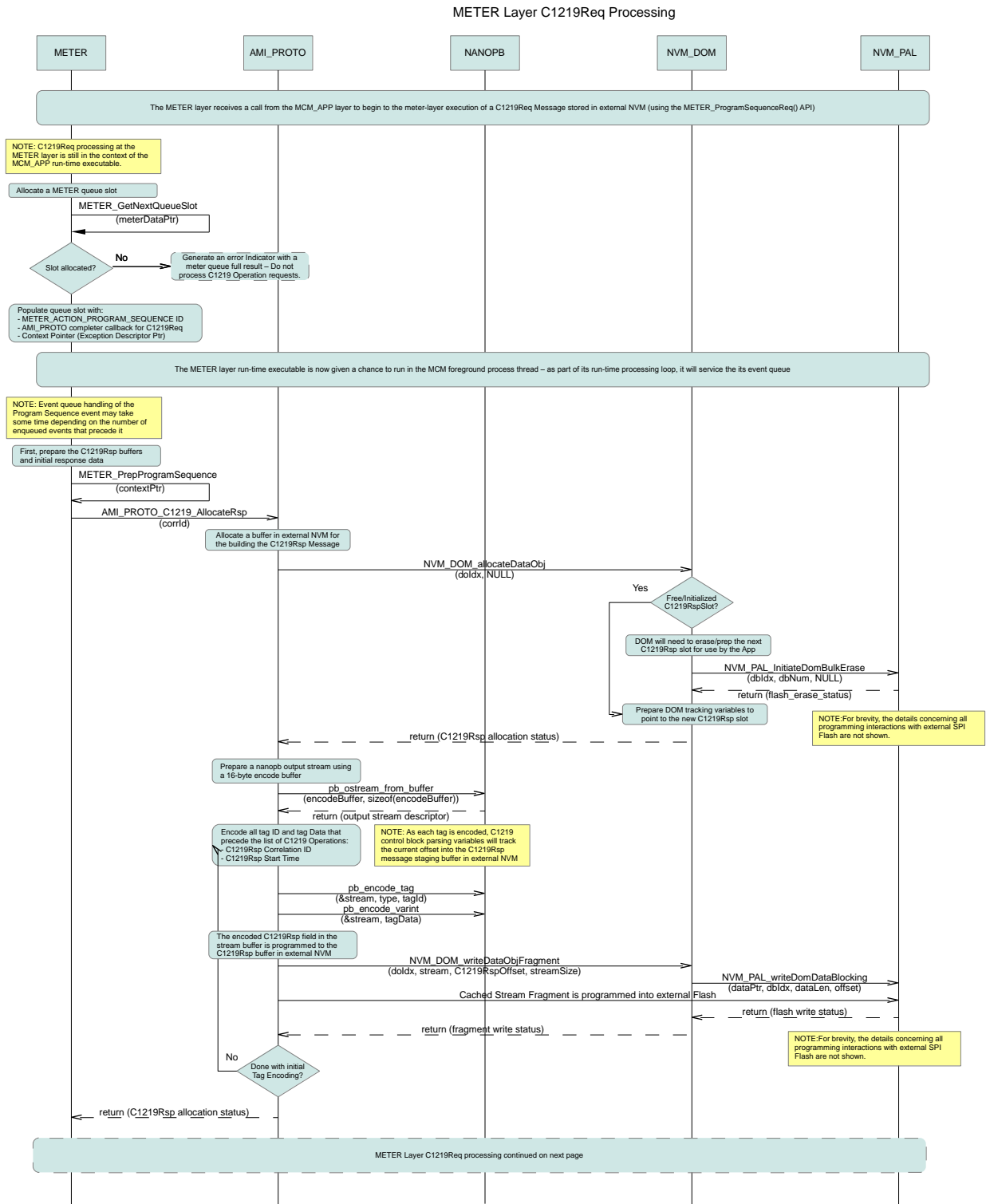
B.2 C1219Req Header Validation

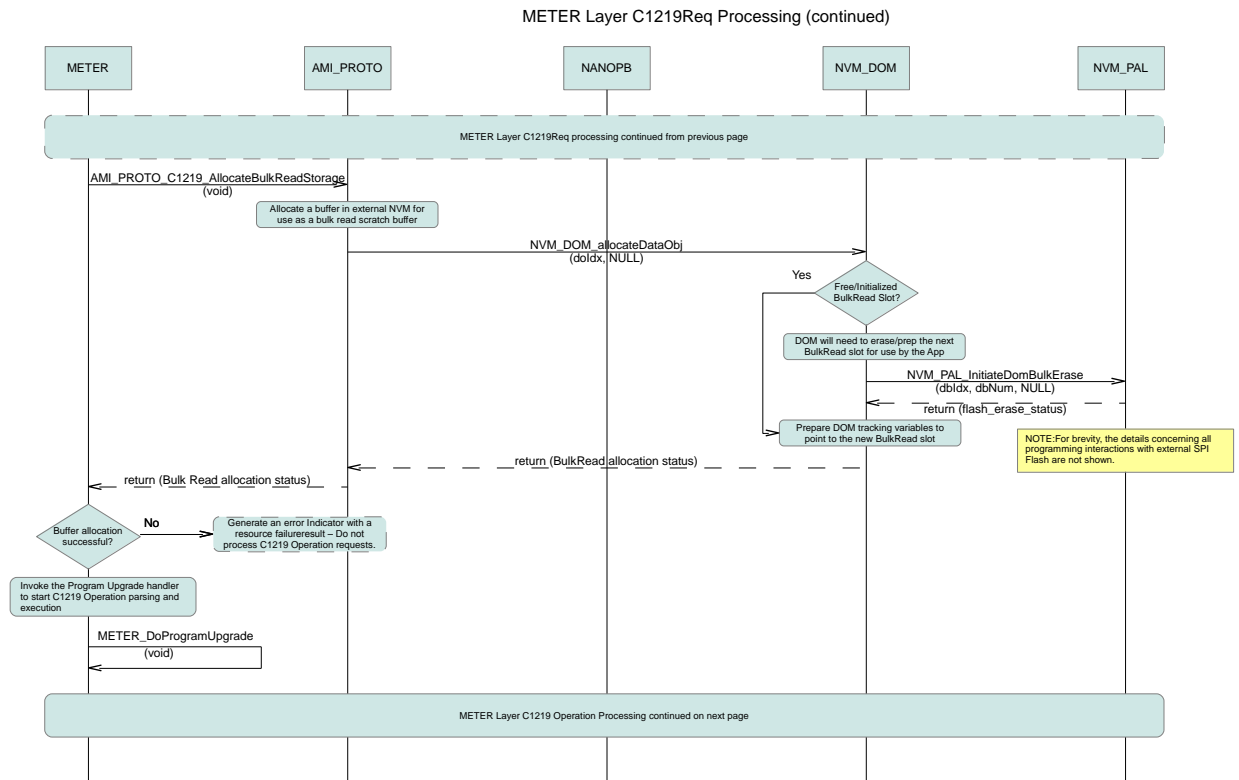


B.3 MCM Application C1219Req Processing

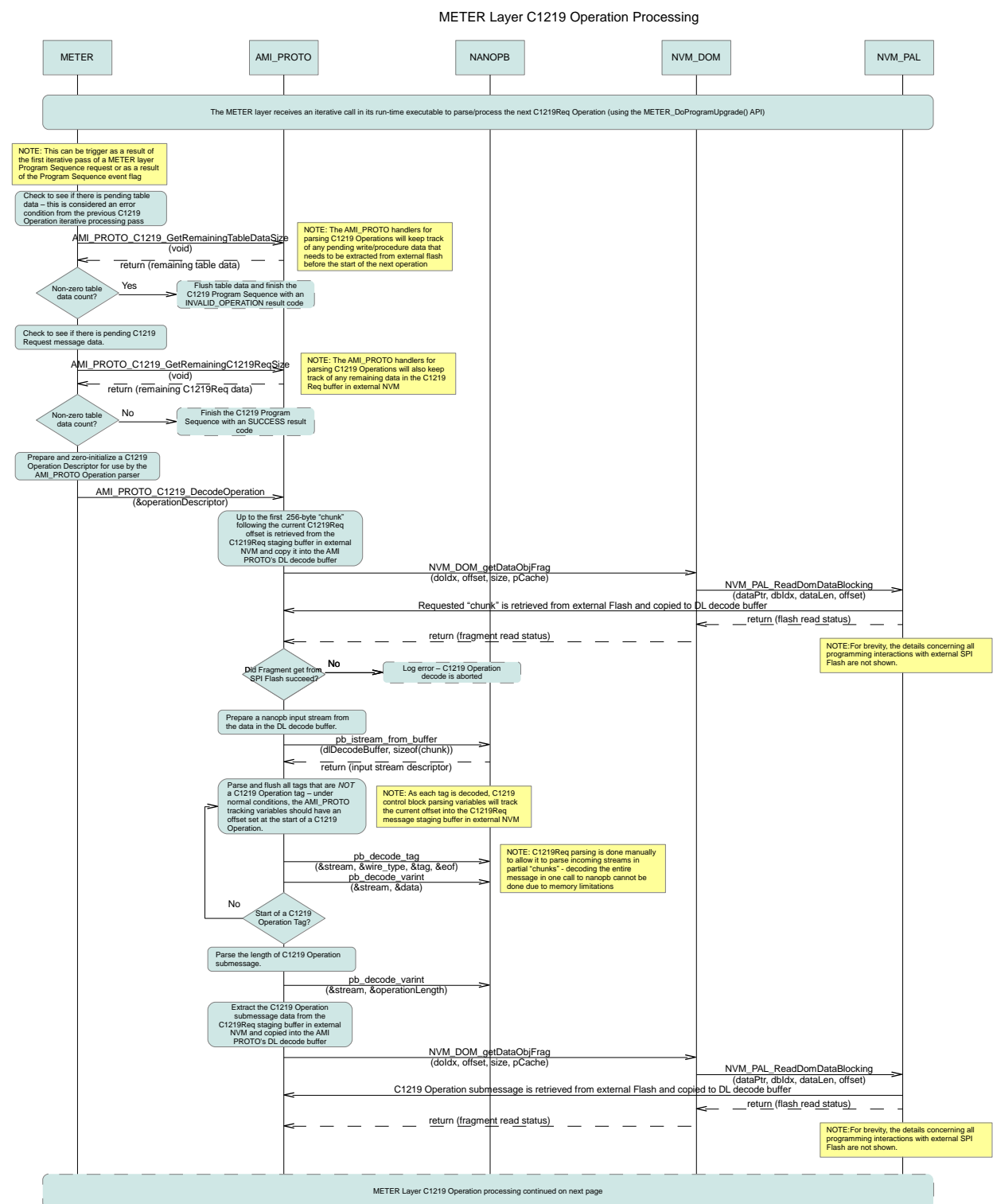


B.4 METER Layer C1219Req Processing

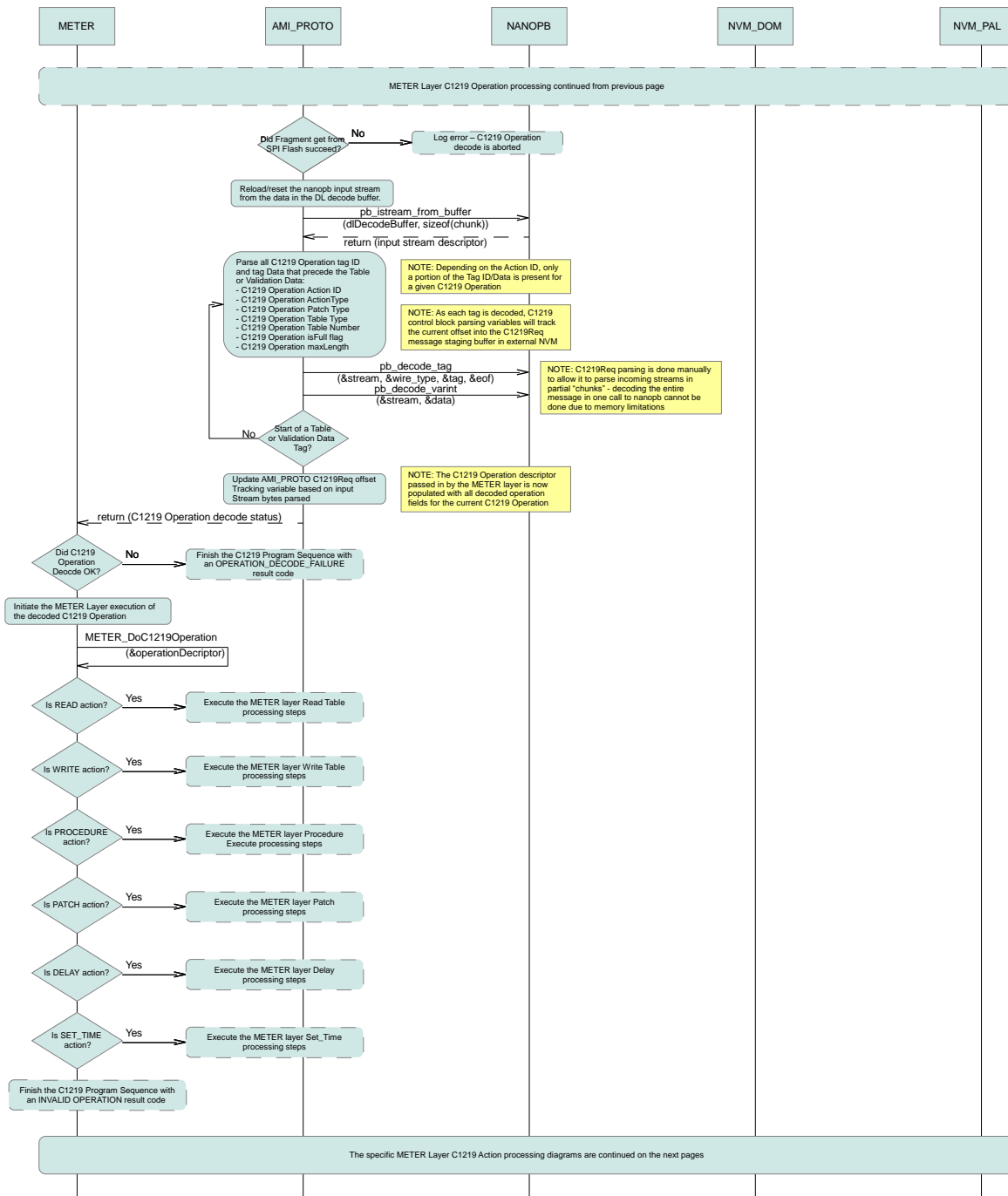




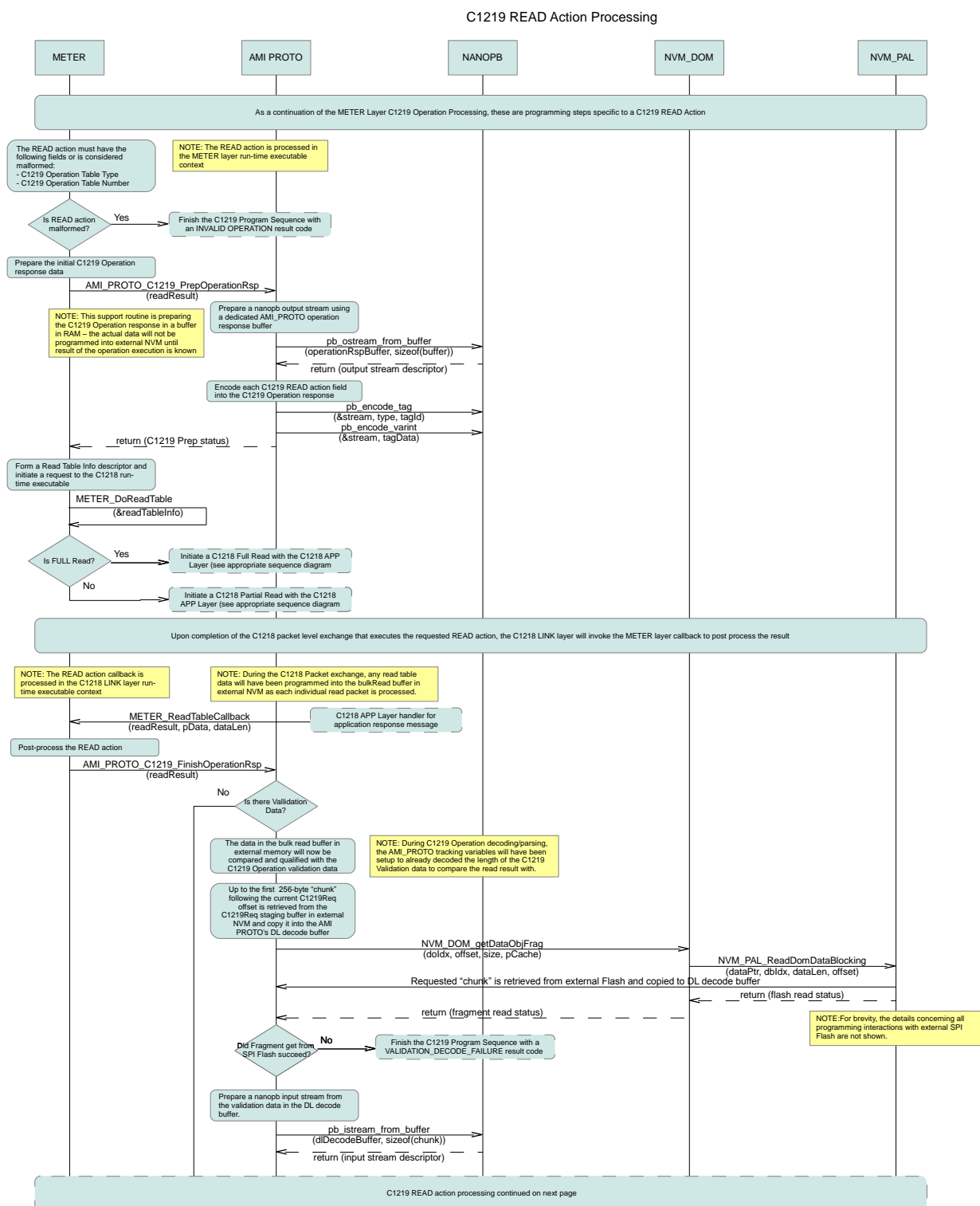
B.5 METER Layer C1219 Operation Processing



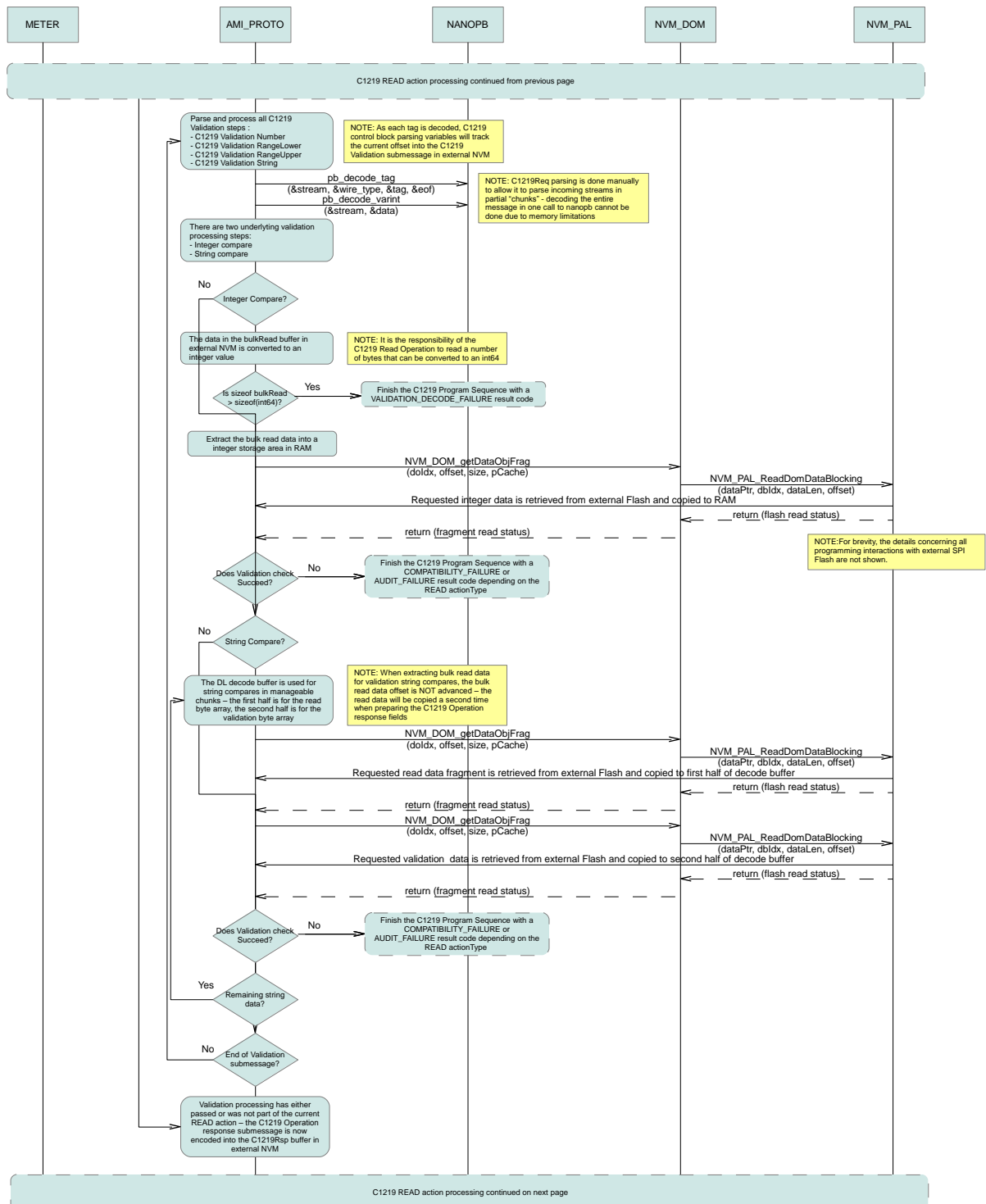
METER Layer C1219Req Processing (continued)



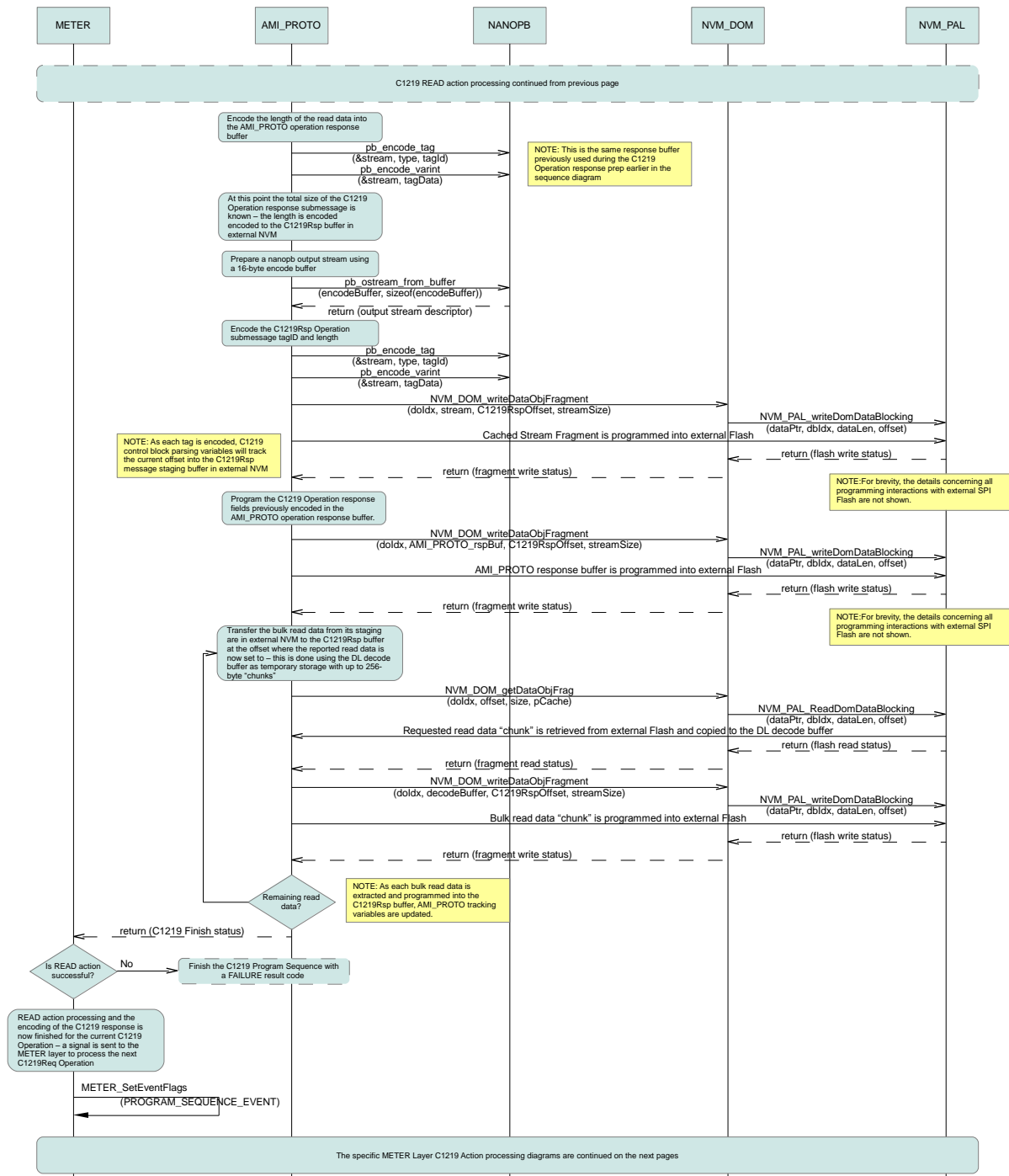
B.6 C1219 READ Action Processing



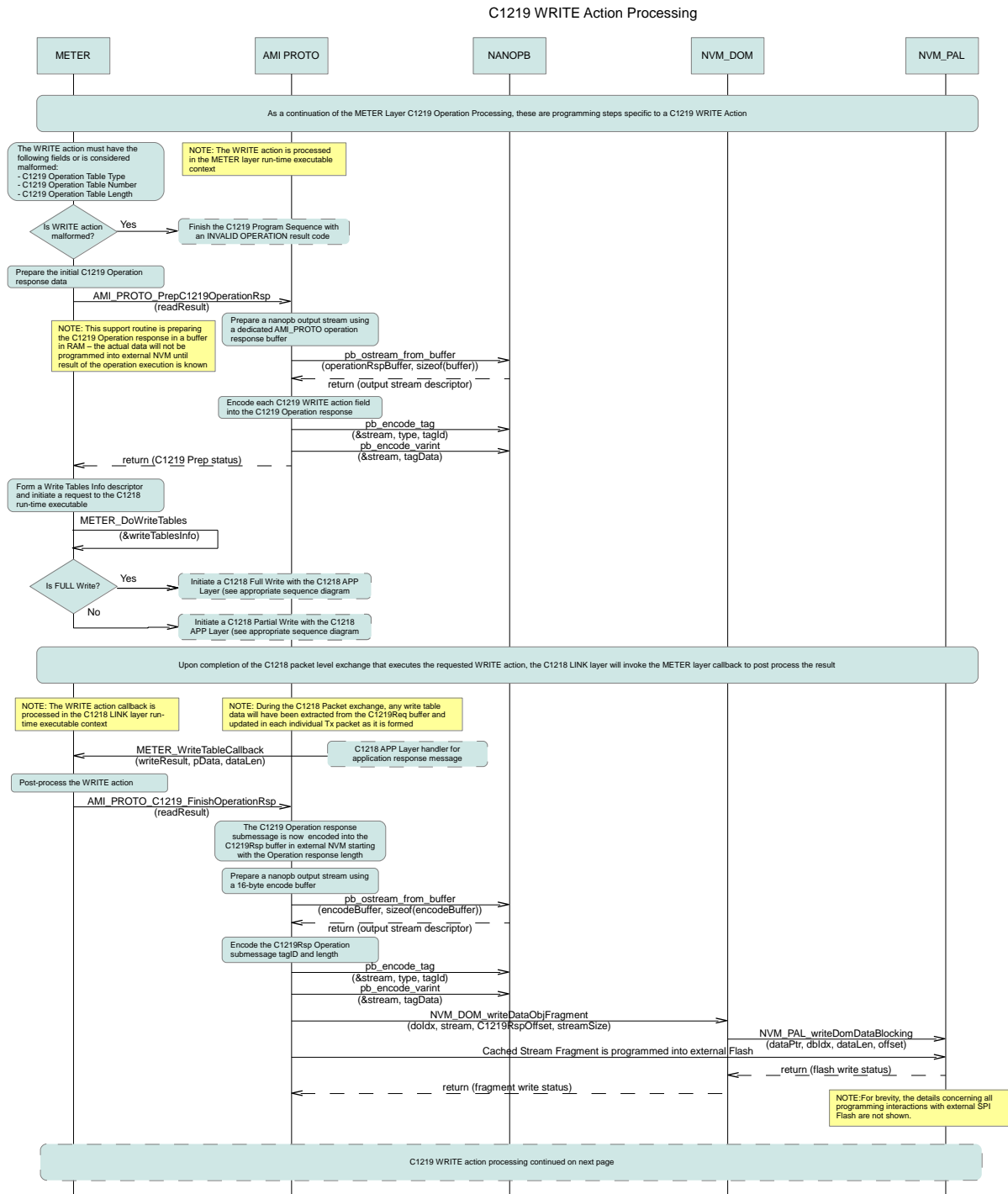
C1219 READ Action Processing (continued)

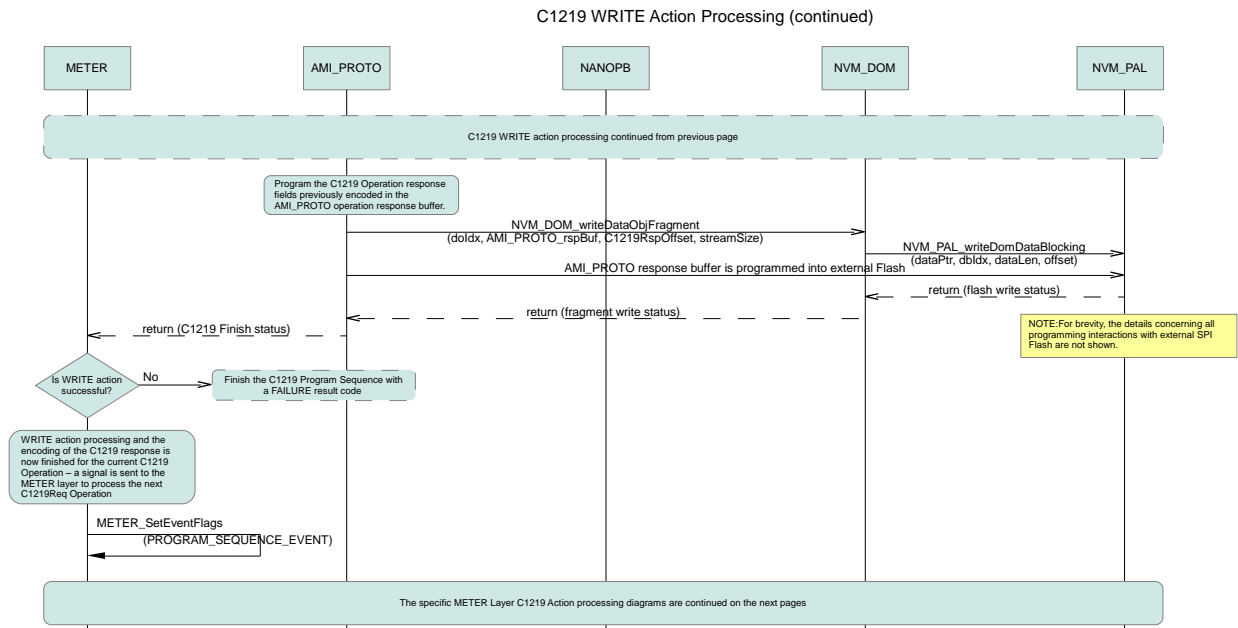


C1219 READ Action Processing (continued)

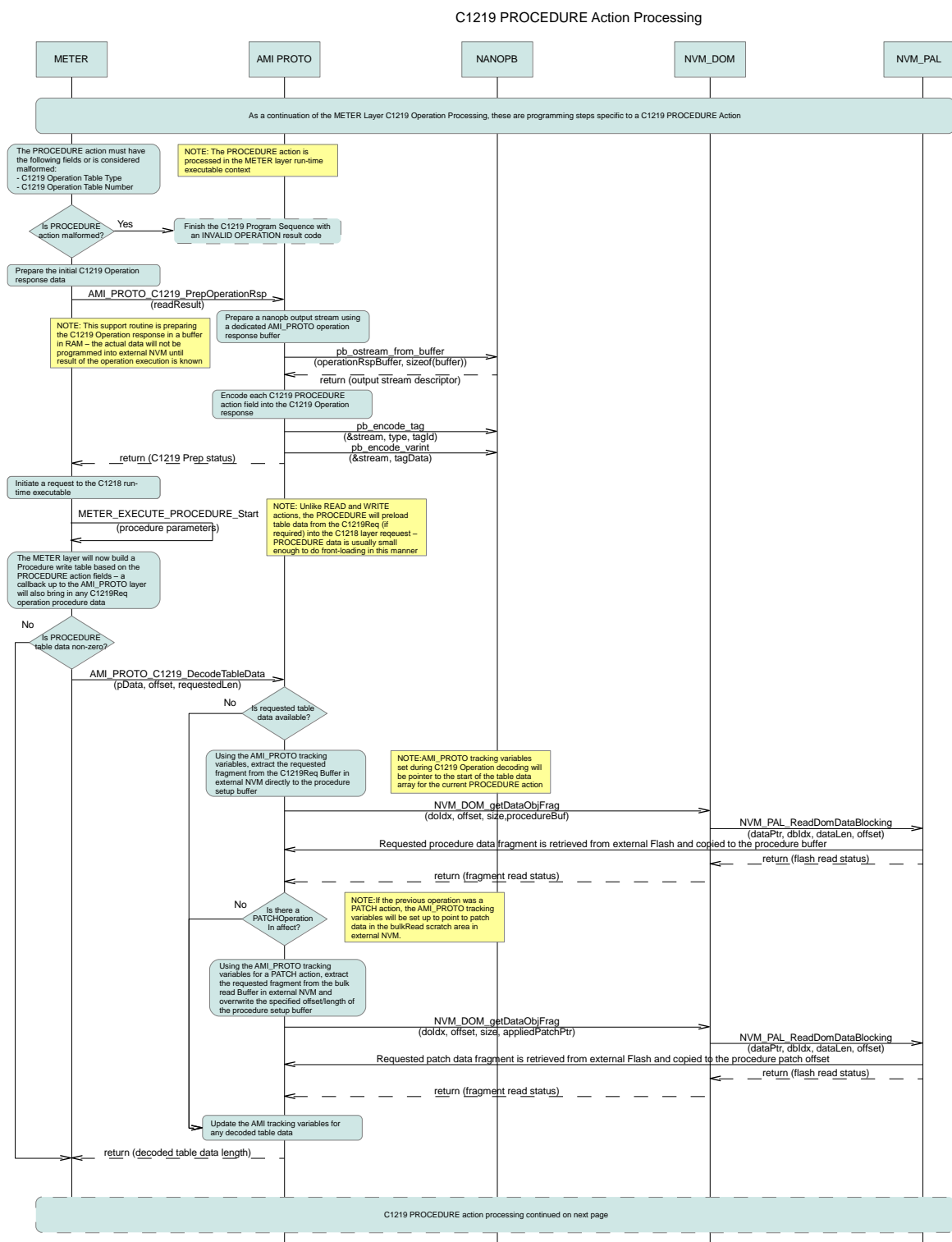


B.7 C1219 WRITE Action Processing

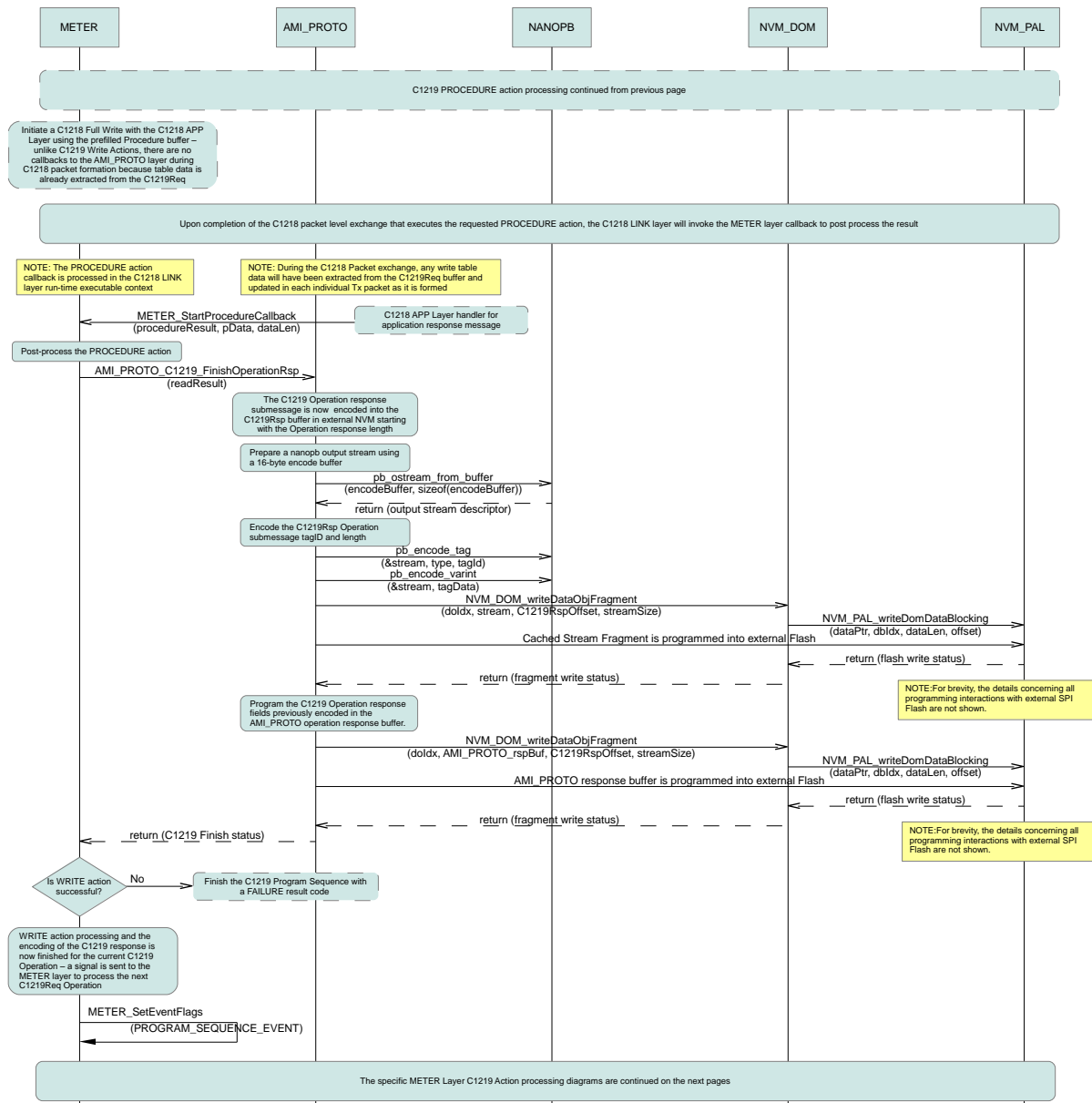




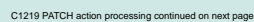
B.8 C1219 PROCEDURE Action Processing

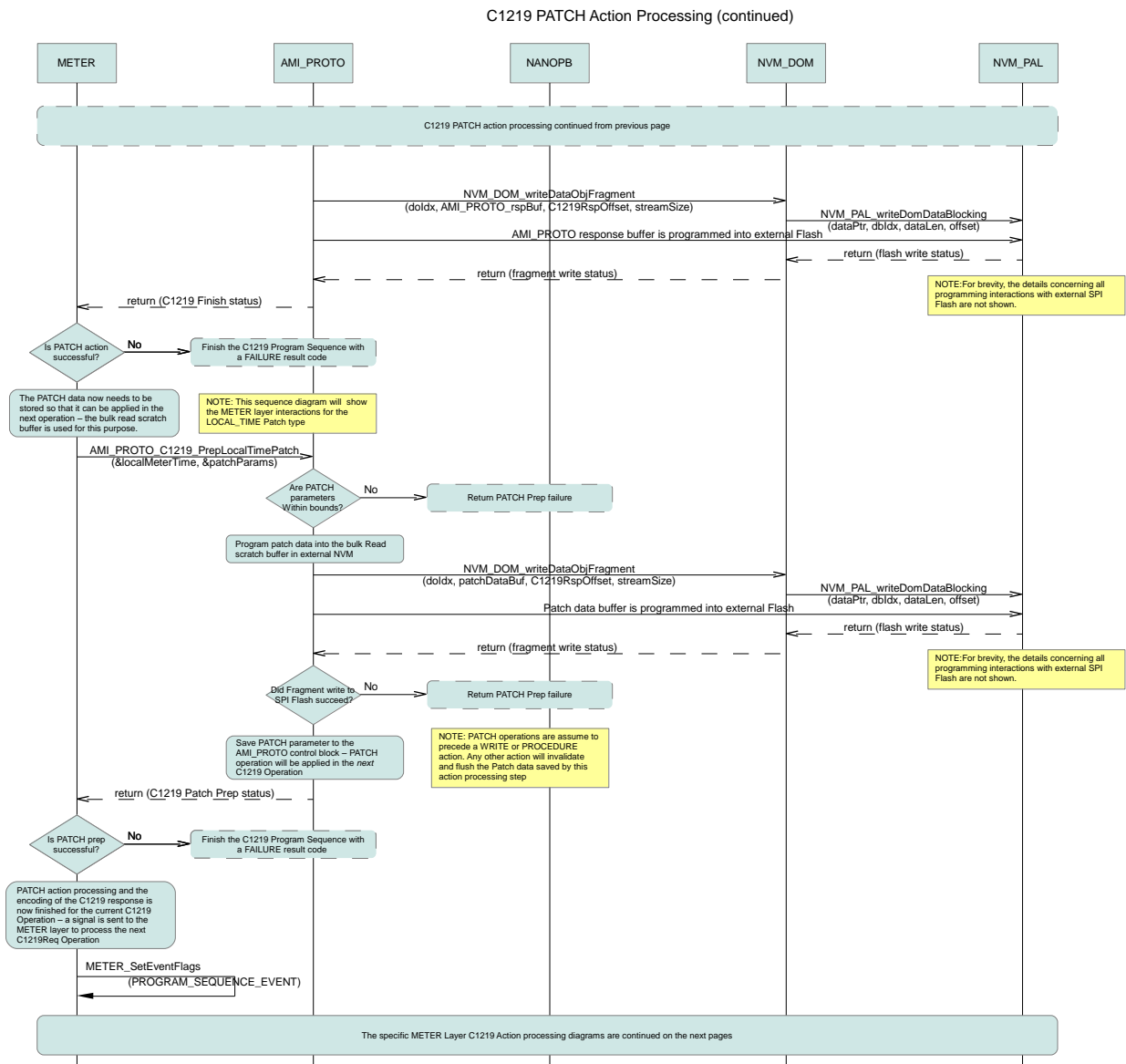


C1219 PROCEDURE Action Processing (continued)

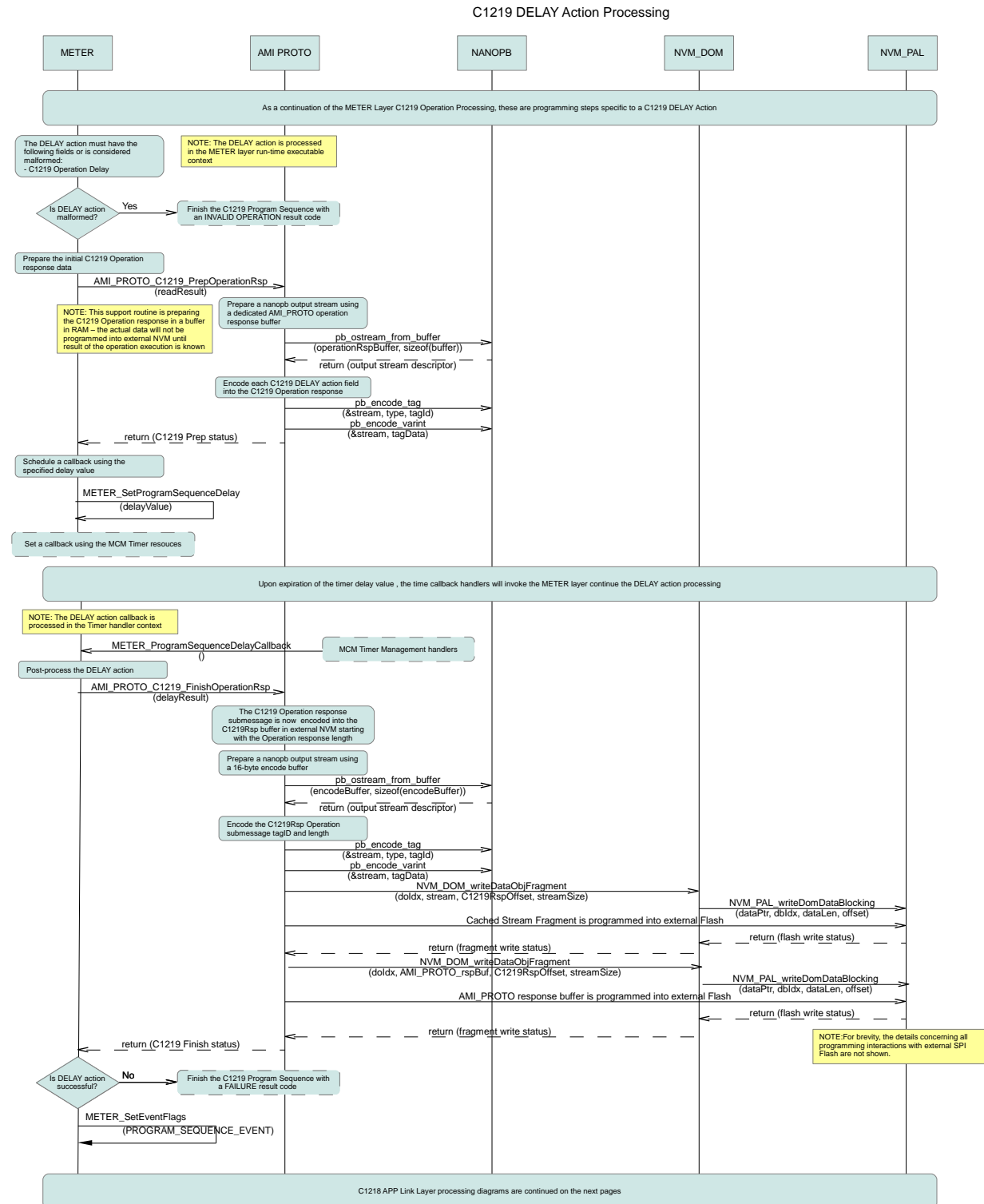


C1219 PATCH Action Processing

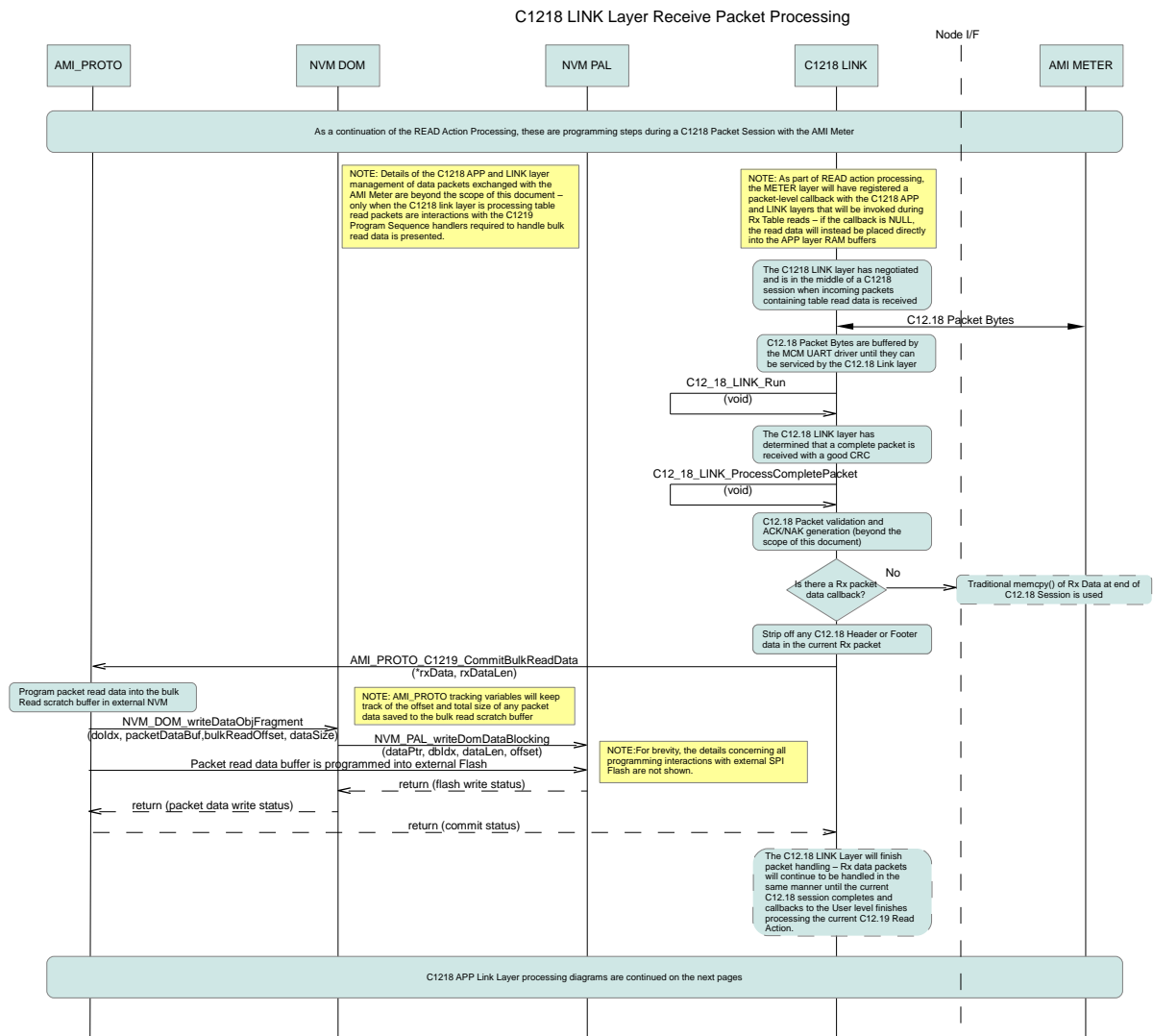




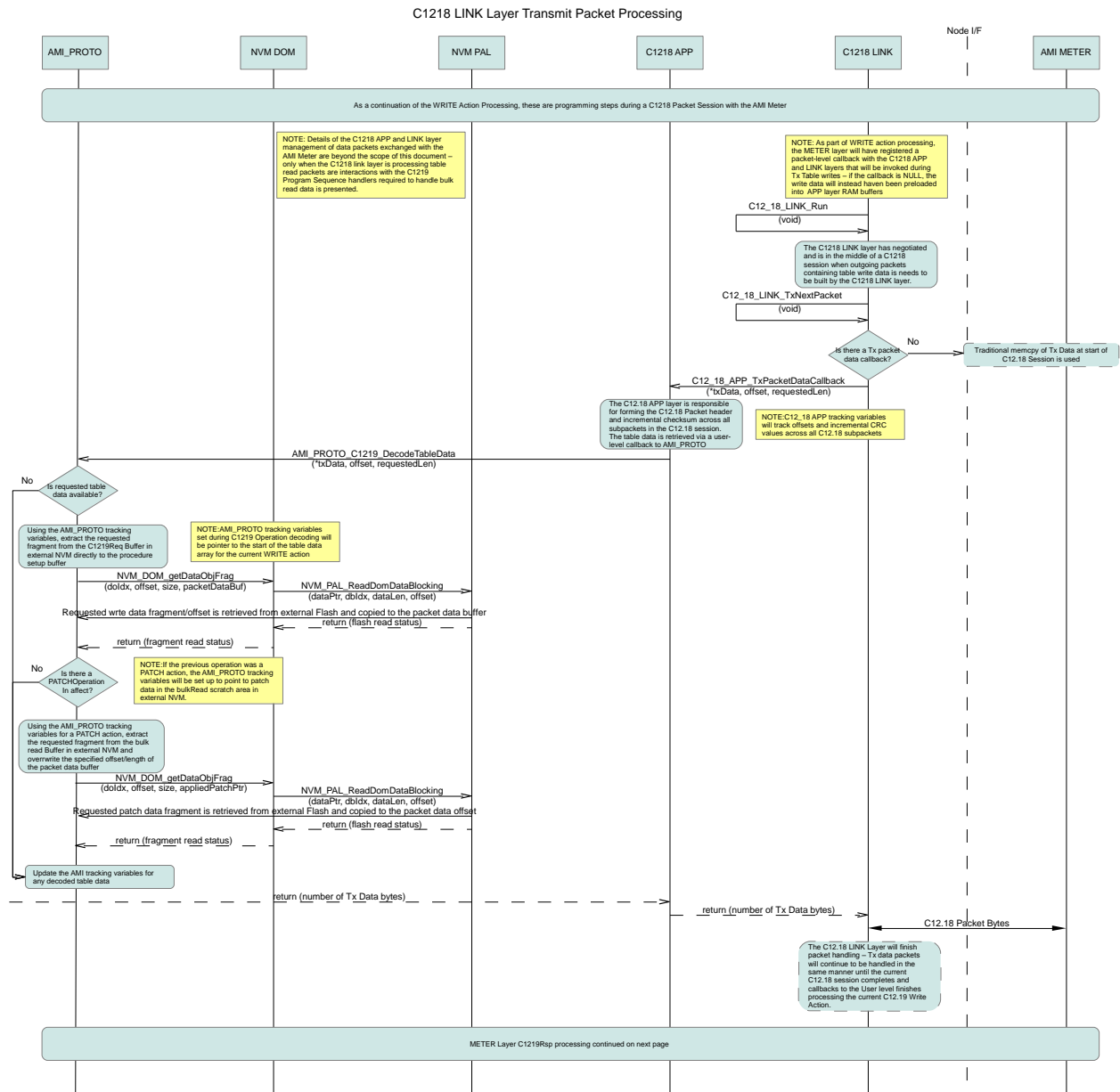
B.10 C1219 DELAY Action Processing



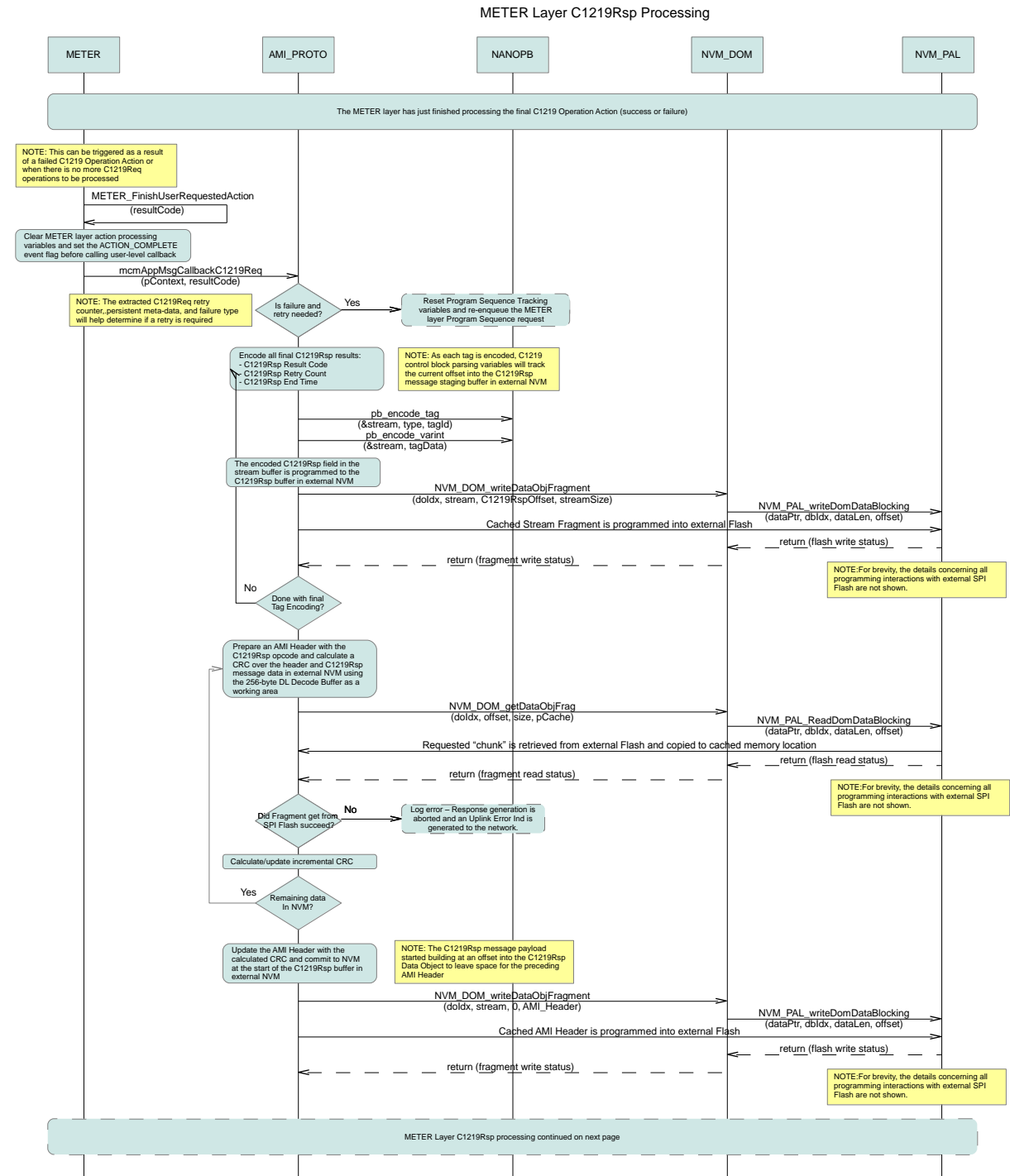
B.11 C1218 LINK Layer Receive Packet Processing

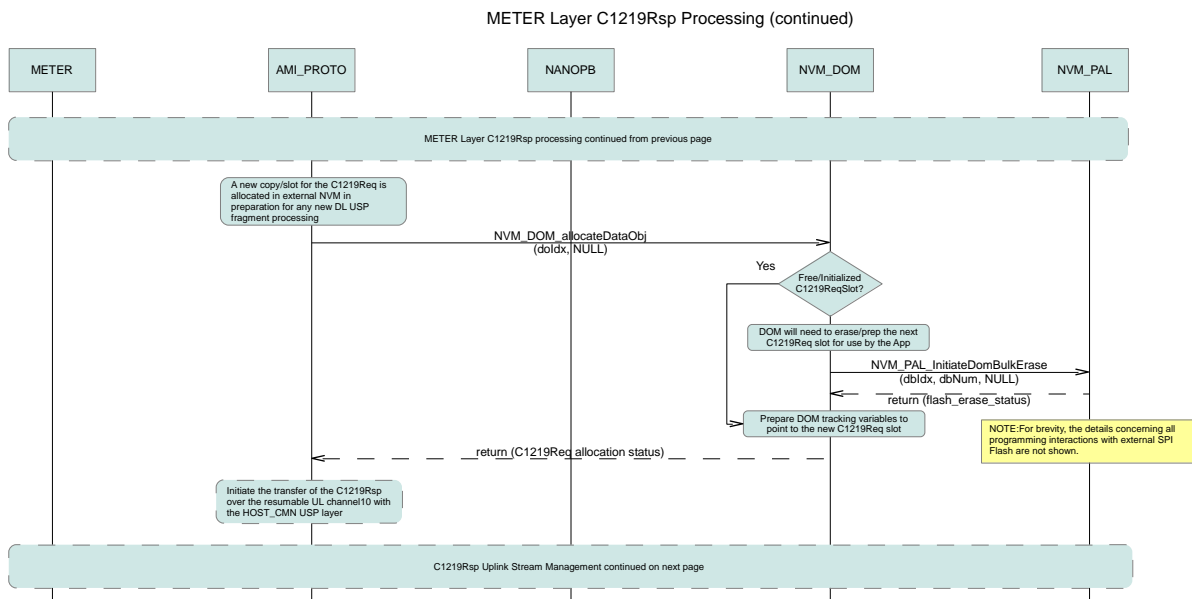


B.12 C1218 LINK Layer Transmit Packet Processing

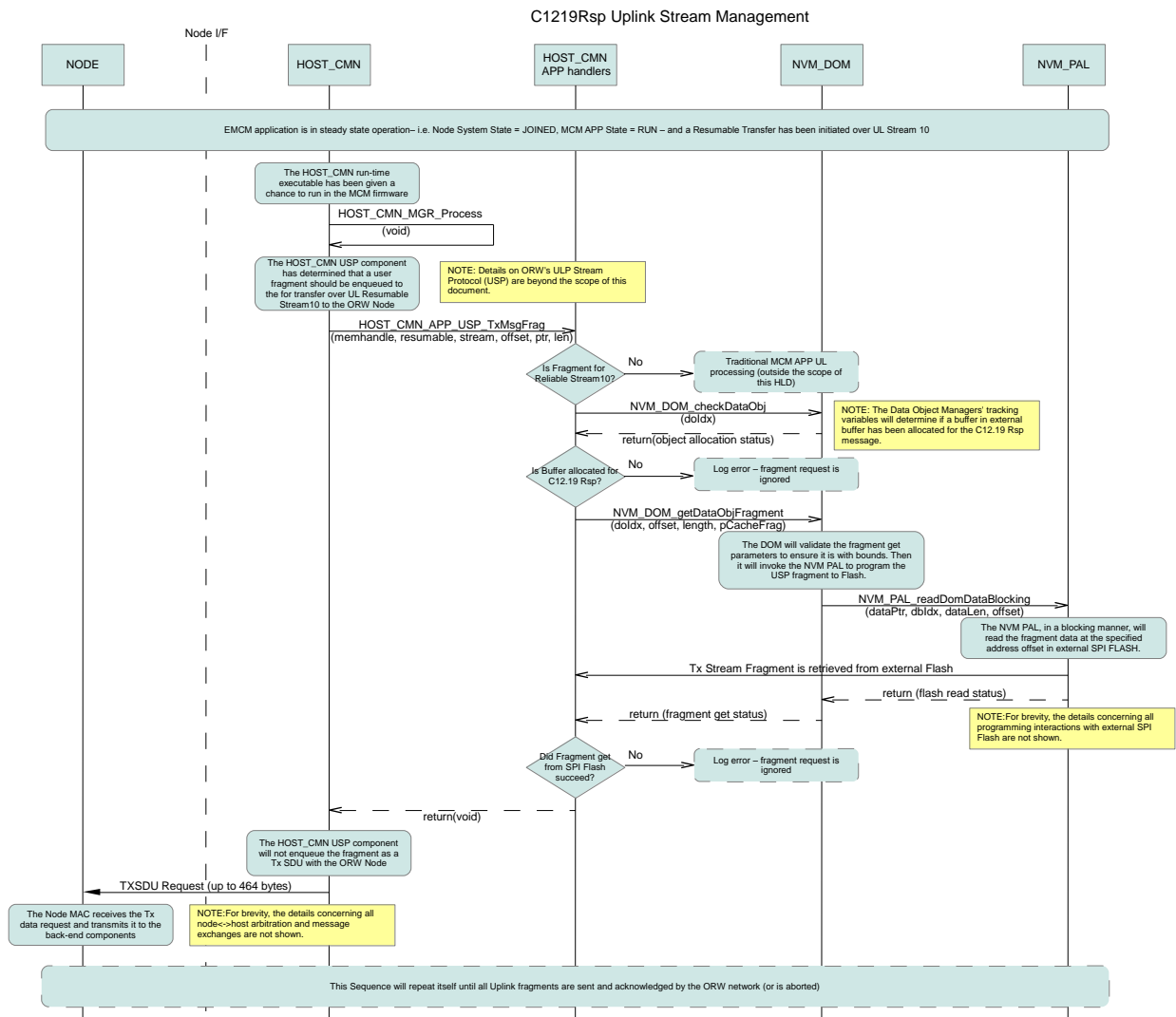


B.13 METER Layer C1219Rsp Processing





B.14 C1219Rsp Uplink Stream Management



Appendix C System Test Outline

This section provides an outline of System Test items that a test engineer/team should focus on to ensure that the C12.19 Program Sequence implementation on MCM meets the specified requirements put forth in the AMI HLD (see reference [1]):

1. OTA C1219 USP Transport Testing

1.1. Point-to-Point C1219Req/Rsp over USP

- 1.1.1. “Happy path” transfer of a C1219Req in a single fragment (i.e., C1219Req length < 458 bytes) – confirm C1219Rsp (preferably with max verbosity).
- 1.1.2. “Happy path” transfer of a C1219Req over two or more single fragments (i.e., C1219Req length > 458 bytes) – confirm C1219Rsp (preferably with max verbosity).
- 1.1.3. “Happy path” transfer of a large C1219Req (i.e., C1219Req length between 10k and 50k) – confirm C1219Rsp (preferable with max verbosity). Additional focus should be placed on the MCM firmware stability during large program sequence execution (i.e., no panic blocks/asserts during transfer).
- 1.1.4. Intermittent jammer tests during multi-fragment C1219Req transfers – confirm all fragments eventually get delivered with a corresponding C1219Rsp.
- 1.1.5. Intermittent power fails/exception tests during multi-fragment C1219Req transfers – after MCM recovers, confirm all fragments eventually get delivered with a corresponding C1219Rsp (assuming recovery occurs before USP timeout).
- 1.1.6. Flush tests during multi-fragment C1219Req transfers – confirm a C1219Req is successfully halted. Follow-up “Happy path” transfers of C1219Req should work without problems.
- 1.1.7. Corrupted transfer of a C1219Req – manipulate the C1219 payload data and/or AMI Header to invalidate C1219Req CRC. Verify integrity error handling for corrupted C1219Req. Follow-up “Happy path” transfers of C1219 should work without problems.

1.2. MUD C1219Req/Rsp over USP

- 1.2.1. Repeat all point-to-point tests with a MUD group of endpoints.
- 1.2.2. Repeat Flush (see test 1.1.6) with a subgroup of MUD endpoints – ensure that unaffected MUD endpoints remain unaffected by the selective “Flush” action.

2. OTA Time Scheduling Testing

2.1. Functional Verification

- 2.1.1. Create a C1219Req without a *startTime* field defined – verify the C1219Req executes immediately after received at the endpoint under test.
- 2.1.2. Create a C1219Req with a *startTime* defined that precedes the current network time as tracked by the endpoint under test – verify the C1219Req executes immediately after received at the endpoint under test.

- 2.1.3. Create a C1219Req with a *startTime* defined for various time offsets at a future network time as tracked by the endpoint under test (e.g., minutes, hours, days) – verify the C1219Req executes at the specified time at the endpoint under test.
- 2.1.4. Create a C1219Req with an *expiryTime* defined that precedes or is set to the current network time as tracked by the endpoint under test – verify that a C1219Rsp is generated and reports an expired time failure code.
- 2.1.5. Create a C1219Req with both a *startTime* and *expiryTime* that are both set at a future network time as tracked by the endpoint under test. However, ensure that the *expiryTime* < *startTime* – verify that a C1219Rsp is generated and reports an expired time failure code at the designated *expiryTime*.

2.2. Exception/adversarial Verification

- 2.2.1. Repeat test for pending *startTime* (see test 2.1.3). Allow a powerfail or other exception reset to occur taking the EMCM offline but allow it to recover before the *startTime* epoch. Verify that the C1219Req executes as the designated *startTime*.
- 2.2.2. Repeat test 2.2.1 but do NOT allow the EMCM to recover until after the designated *startTime*. Verify that the C1219Req executes immediately upon recovery into steady-state operation.
- 2.2.3. Repeat test 2.2.1 but do NOT allow the EMCM to recover until after a designated *expiryTime*. Verify that a C1219Rsp is generated with an expired time failure code upon recovery into steady-state operation.
- 2.2.4. Repeat test for pending *startTime* (see test 2.1.3). Generate a C1219AbortReq and ensure that the C1219Req does NOT execute at the previous expected *startTime*. Verify “Happy path” transfers of C1219Req remain functional after a pending/aborted C1219Req.

3. Verbosity Testing

- 3.1. Set *rspVerbosityType*=1 (*min*) in a C1219Req and initiate a “Happy path” C1219Req transfer, verify that only read operations and final result code are reported in the corresponding C1219Rsp.
- 3.2. Set *rspVerbosityType*=2 (default) in a C1219Req and initiate a “Happy path” C1219Req transfer with one of the C12.19 Operations malformed (e.g., a READ action without a Table Number). Verify that only the failed operation, any read data that precede the failed operation and the final result code are reported in the corresponding C1219Rsp.
- 3.3. Set *rspVerbosityType*=3 (*max*) in a C1219Req and initiate a “Happy path” C1219Req transfer. Verify that ALL C1219Operations, read data, and final result code are reported in the corresponding C1219Rsp.

4. ANSI Meter Compatibility Testing

NOTE: This can easily be done with a simple read/write script that accesses a commonly-used standard FA table (e.g., STD Table#1 – General Manufacturing Identification) and guaranteed to work on any ANSI Meter.

- 4.1. Verify “Happy Path” testing for all EMCM/Meter platforms current support in AMI1.x – each meter/platform should respond with a C1219Rsp (e.g., Raptor, Ptero, Falcon, etc).

- 4.2. Evaluate “Happy Path” testing on an EMCM/Meter platform where the meter is either non-functional or in an otherwise non-operational state. A C1219Rsp should be generated with a C12.18 Link Layer failure code (or Comm Index error).
5. READ Action Testing
 - 5.1. Functional Verification
 - 5.1.1. Verify a C12.19 Read action using offset/length (i.e., PARTIAL) on a STD table. Compare the read data reported in the C1219Rsp with an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT).
 - 5.1.2. Repeat test 5.1.1 on a MFG table specific to the meter under test.
 - 5.1.3. Verify a C12.19 Read action without offset/length (i.e., FULL) on a STD table. Compare the read data reported in the C1219Rsp with an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT). NOTE: The table read may be limited based on the negotiated C12.18 Link Layer limits for number of packets and packet transfer size.
 - 5.1.4. Repeat test 5.1.3 on a MFG table specific to the meter under test.
 - 5.1.5. Repeat tests 5.1.1 through 5.1.4 and include a known validation response string or integer or range of integers (preferably all three). Ensure an actionType is specified as either COMPATIBILITY or AUDIT.
 - 5.1.6. Repeat tests 5.1.1. through 5.1.4 for a sequential series of READS – all valid reads should be accounted for in the C1219Rsp and in the correct order of execution as specified by the C1219Req.
 - 5.1.7. Repeat test 5.1.1 on a large table size (i.e., table size > 1kbytes) – Ensure that all data is accounted for.
 - 5.2. Exception/Adversarial Verification
 - 5.2.1. Repeat test 5.1.1 on a non-existent Read Table – depending on the ANSI Meter handling of a non-supported table, the C1219Rsp may have a zero-length read table or an appropriate error code reported over the C12.18 Link.
 - 5.2.2. Repeat test 5.2.1 except use a table that accesses data that is unsupported by the ANSI Meter under test (e.g., a feature explicitly disabled via soft switched).
 - 5.2.3. Evaluate a C12.19 Read action using missing read values (e.g., a missing table number) – the C1219Rsp should indicate a malformed C1219 Operation error indication.
 - 5.2.4. Evaluate a C12.19 Read action using unnecessary optional values (e.g., a delay value) – the C1219Rsp should disregard the unsupported optional value and execute the C1219 Read action as specified (unless it is otherwise malformed).
 - 5.2.5. Evaluate a C12.19 Read action with additional validation data for a default PROGRAM_AND_READ actionType – the C1219Rsp should indicate a malformed C1219 Operation error indication.
 - 5.2.6. Evaluate other invalid C12.19 Read actions as otherwise would be unsupported without a preceding C12.19 operation access (e.g., certain MFG tables which require a Snapshot PROCEDURE action to access on KVT). This all depends on the creativity and foresight of the systems-level tester.

6. WRITE Action Testing

6.1. Functional Verification

- 6.1.1. Verify a C12.19 Write action using offset/length (i.e., PARTIAL) on a STD table. Follow-up by reading back the data via C1219Rsp or with an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT).
- 6.1.2. Repeat test 6.1.1 on a MFG table specific to the meter under test.
- 6.1.3. Verify a C12.19 Write action without offset/length (i.e., FULL) on a STD table. Follow-up by reading back the data via C1219Rsp or with an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT).
- 6.1.4. Repeat test 6.1.3 on a MFG table specific to the meter under test.
- 6.1.5. Repeat tests 6.1.1 through 6.1.4 and include a preceding PATCH operation. Verify that only the write data specified to be replaced with the PATCH data is overwritten. NOTE: It is recommended that this be used on an innocuous table such a MFG serial ID string that is used for informational purposes – otherwise the PATCH test can be deferred.
- 6.1.6. Repeat tests 6.1.1. through 6.1.4 for a sequential series of WRITES – all valid writes should be accounted for in follow-up reads over the C1219Rsp and in the correct order of execution as specified by the C1219Req.
- 6.1.7. Repeat test 6.1.1 and 6.1.2 on a large table size (i.e., table size > 1kbytes) – Ensure that all data is accounted for in follow-up reads over the C1219Rsp or through use of an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT).

6.2. Exception/Adversarial Verification

- 6.2.1. Repeat test 6.1.1 on a non-existent WriteTable – The C1219Rsp should indicate an appropriate error code reported over the C12.18 Link.
- 6.2.2. Repeat test 6.2.1 except use a table that accesses data that is unsupported by the ANSI Meter under test (e.g., a feature explicitly disabled via soft switched).
- 6.2.3. Evaluate a C12.19 Write action using missing read values (e.g., a missing table number) – the C1219Rsp should indicate a malformed C1219 Operation error indication.
- 6.2.4. Evaluate a C12.19 Read action using unnecessary optional values (e.g., a delay value) – the C1219Rsp should disregard the unsupported optional value and execute the C1219 Write action as specified (unless it is otherwise malformed).
- 6.2.5. Evaluate a C12.19 Write action with a table data size that exceeds the ANSI table data specification.
- 6.2.6. Evaluate other invalid C12.19 Write actions as otherwise would be unsupported without a preceding C12.19 operation access. This all depends on the creativity and foresight of the systems-level tester.

7. Retry Testing

7.1. Functional Verification

- 7.1.1. Specify a non-zero *numRetries* setting in a C1219Req. Whilst in the midst of a normally “Happy path” C1219 Program Sequence execution, interrupt the processing through either

a MCM and/or METER exception event (e.g., remove power, hard reset). Verify that after the MCM re-enters steady-state processing, that the EMCM will re-execute the C1219Req (without any additional user-generated exception).

8. Read-Modify-Write Testing

8.1. Functional Verification

8.1.1. Verify a C12.19 Write-Modify-Bitfield-Write action using offset on a STD table byte that has a programmable bit field. Follow-up by reading back the data via C1219Rsp or with an independent third party utility over the ANSI Meters optical port (i.e., MeterMate or KVT).