Universal Serial Bus Communications Class Subclass Specifications for Network Control Model Devices

Revision 1.1 May 22, 2025

Revision History

Rev	Date	Filename	Comments
1.0	2009-04-30		Initial release
1.0 (Errata 1)	2010-11-24		Merged Errata from "NCMErrata-2010-11-19a.docx"
1.1	2025-05-22		Added NCM 1.1 extended capabilities

Please send comments or questions to: ncm@usb.org

Contributors

Bruce Balden	Belcarra
Stuart Lynne	Belcarra
Alan Berkema	Hewlett Packard
Joel Silverman	K-Micro
Dan Sternglass	MCCI
Peter FitzRandolph	MCCI
Terry Moore	MCCI
Thirumalai Rajan	MCCI
Joe Decuir	MCCI, CSR
David Willcox	Motorola
Ken Taylor	Motorola
Kenji Oguma	NEC
Janne Rand	Nokia
Richard Petrie	Nokia
Tero Soukko	Nokia
Takashi Ninjouji	NTT DoCoMo
Dale Self	Symbian
John Turner	Symbian
Dan Ellis	Synaptics
Morten Christiansen	Synopsys
Saleem Mohammad	Synopsys
Paul E. Berg	USB-IF

NCM 1.1 Contributors

Arnold Liu	Apple
Dan Wilson	Apple
Scott Deandrea	Apple
Avi Shalev	Intel
Dima Ruinskiy	Intel
Reuven Rozic	Intel
Terry Moore	MCCI
Amrutha Chandramohan	Microsoft
Divya Bandaru	Microsoft
Glenn Curtis	Microsoft
Bhavik Thakar	Synaptics
Dan Ellis	Synaptics
Morten Christiansen	Synopsys
Paul E. Berg	USB-IF

Table of Contents

List of figures

List of Tables

1   Introduction

1.1   Purpose

The Communications Class Network Control Model (NCM) Subclass is a protocol by which USB hosts and devices can efficiently exchange Ethernet frames. These Ethernet frames may convey IPv4 or IPv6 datagrams that are transported over a communications network. NCM is intended to be used with high-speed network technologies.

This specification builds upon the USB Communications Device Class subclass specification for Ethernet Control Model devices [USBECM12], with improvements to support much higher data rates:

- Multiple Ethernet frames can be aggregated into single USB transfers.

- In order to minimize overhead when processing the Ethernet frames within the USB device, NCM functions can specify their preferences for how Ethernet frames may be best placed within a USB transfer.

- As of version 1.1 of this specification, the NCM function may further perform a number of offloads to speed up the processing of the Ethernet frames by the host software stack. These include VLAN tag insertion and stripping, IP, TCP and UDP checksum insertion and verification, as well as TCP/UDP segmentation and coalescing (Large Send and Large Receive offloads, respectively). The NCM function advertises to the host its capability to perform any valid combination of the above features. The host may then request the NCM function to perform any valid subset of the advertised capabilities, for each individual Ethernet frame.

1.2   Scope

This document specifies a new control and data plane for networking devices, as a subclass based on the Universal Serial Bus Class Definitions for Communications Devices specification [USBCDC12].   It supports Ethernet [IEEE802.3] and similar networking techniques.

This specification defines the following material applicable to NCM functions:

- The class-specific contents of standard descriptors.

- Additional class-specific descriptors.

- Required interface and endpoint structure.

- Class-specific commands.

- Class-specific notifications.

- The format of data that is exchanged with the host.

- The required behavior.

Behavior that is required of all USB devices and hosts is defined by [USBCORE]. Behavior that is required of all Communications devices is defined by [USBCDC12]. Some commands and notifications are based on material defined in [USBECM12].

1.3   Other USB Networking Specifications

At time of writing, three other USB networking subclasses were defined:

Ethernet Control Model (ECM) [USBECM12]

Ethernet Emulation Model (EEM) [USBEEM10]

ATM Networking Control Model [USBATM12]

ECM and NCM are both applicable to IEEE 802.3 type Ethernet networking functions that can carry IP traffic to an external network. ECM was designed for USB full speed devices, especially to support DOCSIS 1.0 Cable Modems. Although ECM is functionally complete, it does not scale well in throughput or efficiency to higher USB speeds and higher network speeds. NCM draws on the experience gained from ECM implementations, and adjusts the data transfer protocol to make it substantially more efficient.

EEM is intended for use in communicating with devices, using Ethernet frames as the next layer of transport. It is not intended for use with routing or Internet connectivity devices, although this use is not prohibited.

[USBATM12] is applicable to USB-connected devices like ADSL modems which expose ATM traffic directly. Rather than transporting Ethernet frames, [USBATM12] functions send and receive traffic that is broken up into ATM cells and encoded using AAL-2, AAL-4 or AAL-5. Although some of the insights that led to the design of NCM came from experience with [USBATM12] implementations, the two specifications are addressing substantially different target applications.

1.4   Editorial Notes

In some cases material from [USBCDC12] or [USBECM12] is repeated for clarity. In such cases, [USBCDC12] or [USBECM12] shall be treated as the controlling document, unless a change is specifically indicated by this NCM specification.

The NCM specification may be implemented by devices conforming explicitly to [USB20], [USB3] or [USB4] revisions of the USB core specification. For brevity, this document will sometimes refer to [USBCORE] to denote any of the above revisions, which is applicable to the particular implementation.

In this specification, the word 'shall' is used for mandatory requirements, the word 'should' is used to express recommendations and the word 'may' is used for options.

1.5   Related Documents

[IEC60027-2]	IEC 60027-2, Second edition, 2000-11, *Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*
[IEEE802.1Q]	ISO/IEC 8802-1Q (ANSI/IEEE Std 802.1Q): Information technology — Local and metropolitan area networks — Bridges and Bridged Networks, 2022
[IEEE802.3]	ISO/IEC 8802-3 (ANSI/IEEE Std 802.3): Information technology — Local and metropolitan area networks — Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, 2021
[IEEE802.3AZ]	ANSI/IEEE Std 802.3az: IEEE802.3 Amendment 5: Media Access Control Parameters, Physical Layers, and Management Parameters for Energy-Efficient Ethernet, 2010
[RFC768]	RFC 768, User Datagram Protocol, Information Sciences Institute, 1980
[RFC791]	RFC 791, Internet Protocol, Information Sciences Institute, 1981

[RFC826]	RFC 826, An Ethernet Address Resolution Protocol, Network Working Group, 1982
[RFC1034]	RFC 1034, Domain Names - Concepts and Facilities, Network Working Group, 1987
[RFC1035]	RFC 1035, Domain Names - Implementation and Specification, Network Working Group, 1987
[RFC4861]	RFC 4861, Neighbor Discovery for IP version 6 (IPv6), Network Working Group, 2007
[RFC6762]	RFC 6762, Multicast DNS, Internet Engineering Task Force (IETF), 2013
[RFC8200]	RFC 8200, Internet Protocol, Version 6 (IPv6) Specification, Internet Engineering Task Force (IETF), 2017
[RFC9293]	RFC 9293, Transmission Control Protocol (TCP), Internet Engineering Task Force (IETF), 2022
[USB20]	Universal Serial Bus Specification, revision 2.0. **http://www.usb.org**
[USB3]	Universal Serial Bus Specification, revision 3.2. **http://www.usb.org**. Unless otherwise specified, any reference to [USB3] includes [USB20] by reference, especially when referring to full- and high-speed devices.
[USB4]	USB4 Specification, revision 2.0. **http://www.usb.org**. Unless otherwise specified, any reference to [USB4] includes [USB3] by reference, as USB 3.0 tunneling is a requirement of USB4. Note that NCM 1.1 can be used with Gen T as defined in [USB4].
[USBATM12]	USB Subclass Specification for ATM Control Model, Revision 1.2 **http://www.usb.org**
[USBCCS10]	Universal Serial Bus Common Class Specification, revision 1.0. **http://www.usb.org**
[USBCORE]	Used for brevity to denote the appropriate core specification for the device implementation. May be [USB4], [USB3] or [USB20].
[USBCDC12]	Universal Serial Bus Class Definitions for Communications Devices, Revision 1.2. **http://www.usb.org**.
[USBECM12]	USB Subclass Specification for Ethernet Control Model, Revision 1.2 **http://www.usb.org**
[USBEEM10]	USB Subclass Specification for Ethernet Emulation Model, Revision 1.0 **http://www.usb.org**
[USBIAD]	USB Interface Association Descriptor Device Class Code and Use Model, Revision 1.0 **http://www.usb.org**
[USBWMC11]	Universal Serial Bus Subclass Specification for Wireless Mobile Communications, Version 1.1. **http://www.usb.org**

1.6   Terms and Abbreviations

Term	Description
802.3	Second generation networking cabling and signaling, commonly known as Ethernet II.(See [IEEE802.3])
ARP	Address Resolution Protocol, used to map IPv4 addresses to 802.3 MAC addresses. [RFC-826]
Communications Interface	A USB interface that has *bInterfaceClass* set to the class code defined for Communications Class. (See [USBCDC12])
Composite Device	A device or peripheral that exposes two or more functions, each such function being associated with one or more USB Interfaces.
Data Interface	A USB interface that has *bInterfaceClass* set to the class code defined for Data Class. (See [USBCDC12])
Datagram	A collection of bytes forming a single item of information, passed as a unit from source to destination.
DNS	Domain Name System – a distributed name service for resolving hostnames to IP addresses [RFC-1034, RFC-1035]
Descriptor	Data structure used to describe a USB device capability or characteristic.
Device	A logical or physical entity that receives a device address during enumeration.
Ethernet frame	Generic term representing the various kinds of datagrams that may be exchanged over DIX or 802.3 networks.
Function	A collection of one or more interfaces in a USB device, which taken together present a specific capability of the device to the host.
Gbps	Gigabits (10^9 bits) per second
GiB	Gigabinary Bytes: 2^{30} bytes [IEC60027-2]

Term	Description
IPv4	Internet Protocol version 4, a network layer (layer 3) protocol with 32-bit addresses. [RFC-791]
IPv6	Internet Protocol version 6, a network layer (layer 3) protocol with 128-bit addresses. [RFC-8200]
Kbps	Kilobits (10^3 bits) per second
KiB	Kilobinary bytes, 1024 bytes [IEC60027-2]
LAN	Local Area Network, e.g. [IEEE802.3].
Mbps	Megabits (10^6 bits) per second
mDNS	Multicast DNS – a hostname-to-IP address resolution protocol for small networks that do not include a local name server. [RFC-6762]
MiB	Megabinary bytes, 2^{20} bytes [IEC60027-2]
MSS	Maximum Segment Size. Refers to the maximum user payload size (excluding headers) associated with a single MTU-sized frame.
MTU	Maximum Transmission Unit. Refers to the maximum size of Ethernet frame that may be transmitted on a given network. See definition of *wMaxSegmentSize*.
NCM Communications Interface	A Communications Interface with *bInterfaceSubclass* set to the code defined in Table 4-1.
NCM Data Interface	A Data Interface that is identified as a subordinate interface in a UNION descriptor that is associated with an NCM Communications Interface.
NDP	NCM Datagram Pointer: NTB structure that delineates Datagrams (typically Ethernet frames) within an NTB, see Table 3-3. Depending on the NTB format, an NDB may use 16-bit or 32-bit offsets.
NDP Entry	Data structure in an NDP, giving the offset and the length of a single datagram. See section 3.3.
NDPX	Extended NCM Datagram Pointer for NCM 1.1, see Table 4-3 and Table 4-6.
NS	Neighbor Solicitation (Neighbor Discovery) protocol, used to map IPv6 addresses to 802.3 MAC addresses. [RFC-4861]
NTB	NCM Transfer Block: a data structure for efficient USB encapsulation of one or more datagrams. Each NTB is designed to be a single USB transfer. See Figure 3-1 and Figure 3-2.
NTB Format	NTBs have two formats. A "16 bit NTB" primarily uses fields that are 16 bits wide; therefore, such an NTB is limited to a maximum size of 64 KiB. A "32-bit NTB" primarily uses fields that are 32 bits wide; therefore, a 32-bit NTB may be as long as 4 GiB.
NTBX	Extended NCM Transfer Block for NCM 1.1: a data structure for efficient USB encapsulation of one or more datagrams. Each NTBX is designed to be a single USB transfer. See Figure 4-1 and Figure 4-2.
NTH	NTB Header: a data structure at the front of each NTB, which provides the information needed to validate the NTB and begin decoding. Depending on the NTB format, this may be either a 16-bit NTH16 (Table 3-1) or a 32-bit NTH32 (Table 3-2).
NTHX	Extended NTB Header: a data structure at the front of each NTBX, which provides the information needed to validate the NTBX and begin decoding. See Table 4-1.
TCP	Transmission Control Protocol, a transport layer (layer 4) protocol used on top of IP. [RFC-9293]
UDP	User Datagram Protocol, a transport layer (layer 4) protocol used on top of IP. [RFC-768]
Union	A relationship among a collection of one or more interfaces that can be considered to form a functional unit.
VLAN	Virtual Local Area Network, as defined in [IEEE802.1Q]. Defines a system of tagging standard Ethernet frames, allowing network administrators to group together hosts connected to different data link layer (layer 2) networks.
WAN	Wide Area Network (e.g., the Internet)

2   Overview

2.1   General

This subclass specification includes specifications for USB-connected external data network adaptors that model IEEE 802 family Layer 2 networking functionality.

Devices of these subclasses shall conform to:

- [USBCORE] USB Specification, and

- Communications Device Class 1.2 [USBCDC12]

The principal advantage of using NCM lies in its method of transporting multiple datagrams inside single USB bulk transfers. In addition to reducing interrupt overhead, the NCM specification allows the sender of data to arrange the datagrams within the transfer so that the receiver need do minimal copying after receipt.

This specification defines two ways of encapsulating datagrams, one allowing transfers up to 64KiB (up to forty (40) 1514-byte [IEEE802.3] Ethernet frames), and another for transfers of up to 4GiB, supporting thereby [USB20] High Speed, [USB3] SuperSpeed or [USB4] Gen T data rates.

Devices implementing the NCM function may be composite devices as described by [USBWMC11] or [USBIAD].

2.2   The Architecture of NCM Functions

An NCM function is implemented by an NCM Communications Interface and an NCM Data Interface. The NCM Communications Interface is used for configuring and managing the networking function. The NCM Data Interface is used for transporting data, using the endpoints defined by that interface. Generally, the NCM Communications Interface and the NCM Data Interface are managed by a single driver on the USB host. The logical connections between host driver and NCM function are shown in Figure 2-1, and the control and data connections are shown schematically in Figure 2-2.
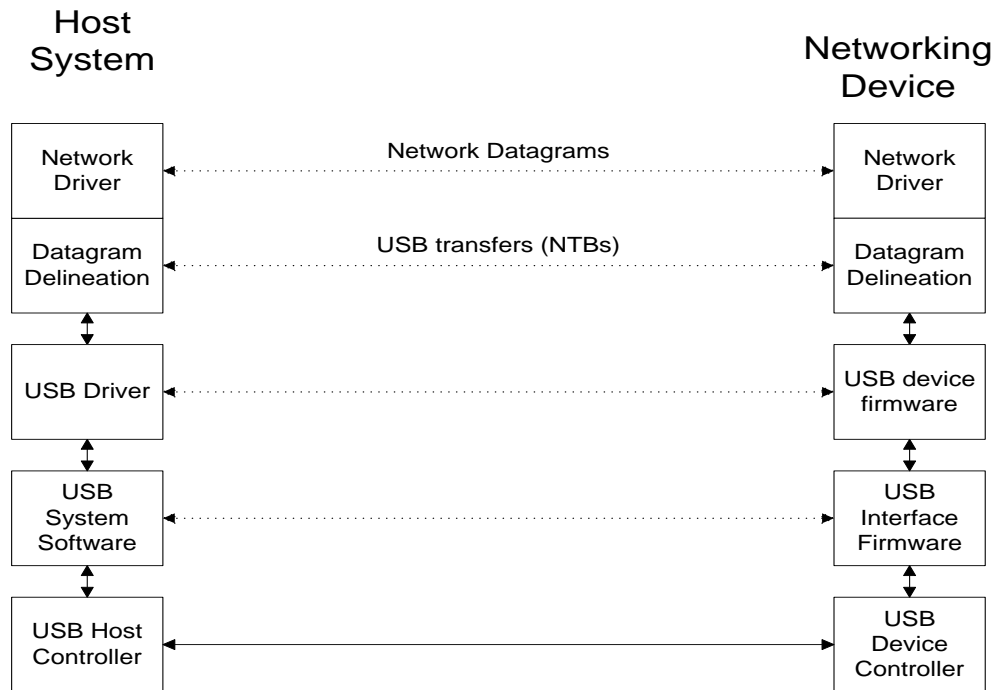
Figure 2-1 - NCM Function Example
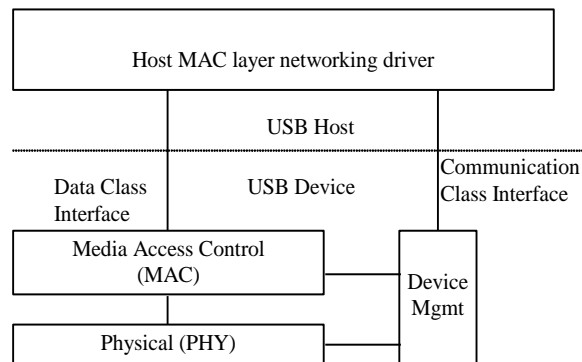
Figure 2-2 - Network Control Model

Although an NCM function may stay in an "always connected" state, some management requests may be required to properly initialize both the function and the host networking stack. There also may be occasional changes of function configuration or state, e.g., adding multicast filters in response to changes in the host networking stack.

2.3   Extended Capabilities

NCM version 1.1 adds additional features to allow NCM to be more useful with faster Ethernet-like interfaces. The specification intends that older host drivers will be able to use newer devices using the basic features of NCM 1.0, and that new host drivers will be able to take advantage of the newer features.

New capabilities in NCM 1.1 include:

1.   Enhanced transmit and receive features, such as: L3/L4 header checksum calculation and verification offloads, Large Send and Large Receive offloads, VLAN tag insertion and stripping, RSS hash computation.

2.   Improved medium handling capabilities, for controlling parameters of specific physical media, such as Flow Control, Energy Efficient Ethernet, etc.

3.   Extended low-power capabilities, including the following:

   • Wakeup on a subset of possible sources, including those previously defined in [USBECM12].

   • Presence offload, whereby the network device may autonomously reply to specific network protocols, and maintain device presence in the network, without waking the host system.

To maintain backward compatibility, NCM 1.1 introduces the concept of extended capabilities. NCM functions may implement none, some, or all of the advanced capabilities introduced in version 1.1. Hosts with enhanced drivers can discover functions with advanced capabilities using the following procedure.

1.   Discover an NCM function using the class and subclass codes from the control interface descriptor or the interface association descriptor of the function.

2.   Check bit D7 of the byte *bmNetworkCapabilities* in the NCM Functional Descriptor. If set, the function supports the NCM 1.1 extended requests (section 8.2). If not set, the host should avoid issuing any of the extended requests, and the response to them is undefined.

3.   If extended capabilities are supported, the host driver may use the *SetExtendedCapabilityMode* command (8.2.1) to switch to NCM 1.1 mode, and may issue the *GetExtendedCapabilities* command (8.2.3) to retrieve the extended capabilities descriptor. The host shall parse the descriptors returned by that request and identify the capabilities supported by the function. The host may safely use the supported capabilities when operating the function.

2.4   NCM 1.0 / 1.1 Compatibility Matrix and State Diagram

NCM 1.1 extended capabilities must be explicitly advertised through bit D7 in *bmNetworkCapabilities* as described above. NCM 1.0 functions shall not set this bit (it is reserved prior to NCM 1.1), and NCM 1.0 hosts, which are not NCM 1.1-aware shall not query for it. This leads to the following compatibility matrix:

Table 2-1: NCM 1.0/1.1 Host/Function Compatibility Matrix

	NCM 1.0 Function	NCM 1.1 Function
NCM 1.0 Host	Operation shall be according to NCM 1.0 specification.	NCM 1.1 function is initialized in NCM 1.0 mode and is fully NCM 1.0 compliant. As the host is not NCM 1.1-aware, the operation shall be according to NCM 1.0 specification.
NCM 1.1 Host	The host may operate in NCM 1.0 mode. This is the recommended approach. Alternatively, the host may declare NCM 1.1 as the minimum required standard and reject the function as non-compliant.	NCM 1.1 function is initialized in NCM 1.0 mode. After verifying NCM 1.1 capability, the host shall explicitly transfer the function to NCM 1.1 mode by issuing the command *SetExtendedCapabilityMode* (8.2.1). Further operation shall be according to NCM 1.1 specification.

NCM 1.1 mode includes changes to both the data path and the control path of the NCM function.

1.   The NCM 1.1 data transport is required to take advantage of some of the optional capabilities the NCM 1.1 function may support, and includes additional changes to simplify both hardware and software operations. The NCM 1.0 data transport and NCM 1.1 data transport are described in Sections 3 and 4, respectively.

2.   The NCM 1.1 control requests are required to configure the extended capabilities, and provide alternate interfaces to some of the basic capabilities already existing in NCM 1.0, as well as in [USBECM12] and [USBCDC12]. NCM 1.0 Descriptors are discussed in Section 6. NCM 1.0 class-specific messages are described in Section 7. NCM 1.1 extended capability descriptors and control requests are described in Section 8. An NCM 1.1 host should use the NCM 1.1 control messages, and may fall back on NCM 1.0 only when the request is revision-agnostic, and when equivalent NCM 1.1 messages are not defined.

3.   An NCM 1.1 function may provide in a single notification the same information for which multiple notifications would be required in NCM 1.0 or [USBECM12]. Refer to sections 7.4 and 8.4 for details on NCM 1.0 and NCM 1.1 notifications, respectively.

The diagram below shows the basic states and state transition of an NCM function. The function is always initialized in alternate setting 0 and NCM 1.0 mode. While in alternate setting 0, an NCM 1.1 function may be set to NCM 1.1 mode through the *SetExtendedCapabilityMode* control request (see 8.2.1). When alternate setting 1 is selected for the data interface, the function shall use either the NCM 1.0 Data Transport (Section 3) or NCM 1.1 Data Transport (Section 4), depending on whether the *SetExtendedCapabilityMode* request has been sent. Refer to 6.3 for additional details.

From any state, resetting the data interface to alternate setting 0 shall return the NCM function to the initial state.
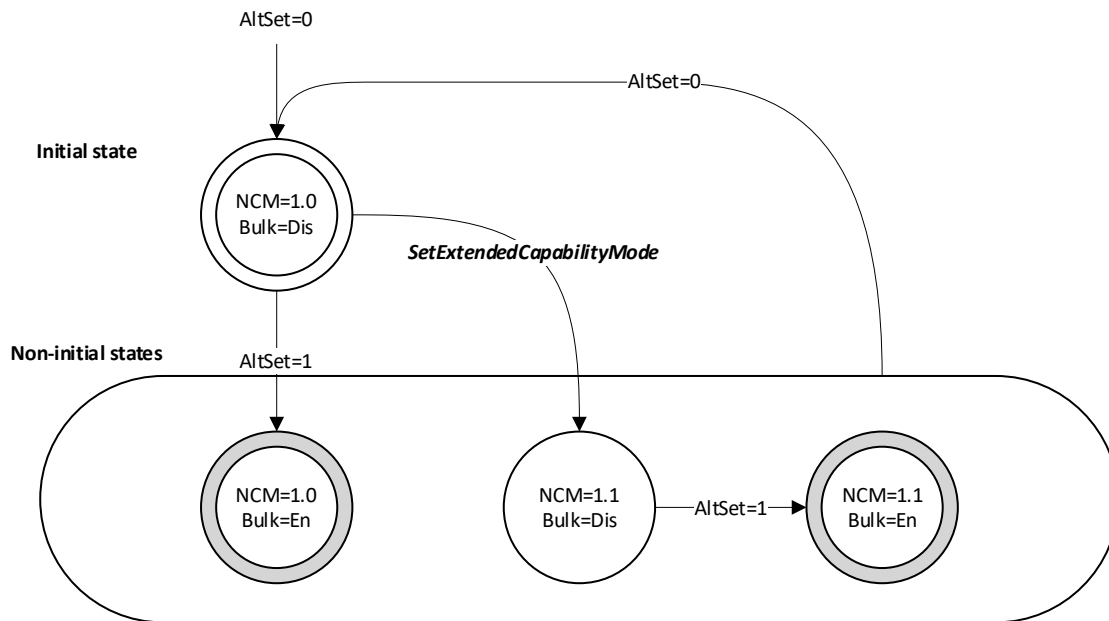
Figure 2-3 – NCM Function State Diagram

2.5   Interactions with other USB core features

1.   An NCM function may be suspended and resumed as defined in [USBCORE].

2.   An NCM function may support remote wakeup as defined in [USBCORE].

3   NCM 1.0 Data Transport

3.1   Overview

Unless explicitly specified otherwise, the information in this section applies only to NCM functions operating in NCM 1.0 mode, as defined in section 2.4. For NCM functions operating in NCM 1.1 mode, refer to section 4.

NCM allows device and host to efficiently transfer one or more Ethernet frames using a single USB transfer.   The USB transfer is formatted as a NCM Transfer Block (NTB).

Figure 3-1 and Figure 3-2 outline the structure of the NTB. Each NTB consists of several components.

- It begins with an NCM Transfer Header ("NTH"). This identifies the transfer as an NTB, and provides basic information about the contents of the NTB to the receiver (3.3.3.1).

- The NTH effectively points to the head of a list of NDP structures (NCM Datagram Pointers). Each NDP in turn points to one or more Ethernet frames encapsulated within the NTB (3.3.3.2).

- Finally, the NTB contains the Ethernet frames themselves (3.3.3.3).

Within any given NTB, the NTH always shall be first; but the other items may occur in arbitrary order.

There are two kinds of NTB. NTBs that are shorter than 65,536 bytes in length can be represented as "sixteen bit NTBs" (NTB-16). NTBs of up to 4 GiB in size can be represented as "thirty-two bit NTBs" (NTB-32). The structures are abstractly the same, but NTB-16 form primarily uses 16-bit fields. The two formats are shown in Figure 3-1 and Figure 3-2, respectively.

Although two formats are defined, a function only uses one format or the other at a given time. The same format is used for IN and OUT transfers. The host selects the format to be used.

NTBs cannot be of arbitrary size; functions normally advertise their upper limits to the host. NCM allows functions to have different maximum NTB sizes for transmit and receive. The sender of an NTB may vary the size of NTBs as needed.

Header		4 bytes: 'NCMH' with 'N' in first byte
	Sig(NTH16)	
	len(header)	2 bytes: 0x000C in little-endian
	seq	2 bytes: incrementing sequence number reset by NCM reset (select alt-setting 0)
Datagram[0]	L	2 bytes: length L
	index(NDP)	2 bytes: offset of first NDP from start of xfr – must be multiple of 4

	Sig(NDP16)	4 bytes: 'NCMx' w/ 'N' in first byte
Datagram[1]	len(NDP header)	2 bytes: includes sig. (size must be 8+k*4, k>1) eg 16, 20, etc.
	index(next NDP)	2 bytes: link to next NDP16, or zero if none
Datagram[2]	index(Datagram[0])	2 bytes: offset from byte 0 of hdr
	len(Datagram[0])	2 bytes: length, zero ➔ end
	index(Datagram[1])	Repeat if/as needed
NDP[0]	len(Datagram[1])	
NDP[0] may be anywhere in the transfer buffer, often will actually be right after header		Zero padding may be inserted here if convenient
	0	Required termination of header
	0	

L bytes

Figure 3-1 - NTB layout details (16 bit)

Header		4 bytes: 'ncmh' with 'n' in first byte
	Sig(NTH32)	
	len(header)	2 bytes: 0x0010 in little-endian
	seq	2 bytes: incrementing sequence number reset by NCM reset (select alt-setting 0)
Datagram[0]	L	4 bytes: length L
	index(NDP)	4 bytes: offset of first NDP32 from start of xfr – must be multiple of 4

	Sig(NDP32)	4 bytes: 'ncmx' w/ 'n' in first byte
Datagram[1]	len(NDP)	2 bytes: includes sig. (size must be 16+k*8, k>1) eg 32, 40, etc.
	Reserved (0)	2 bytes of padding
Datagram[2]	index(next NDP)	4 bytes: link to next NDP32, or zero if none
	Reserved (0)	4 bytes of padding
	index(Datagram[0])	4 bytes: offset from byte 0 of hdr
NDP[0]	len(Datagram[0])	4 bytes: length
NDP[0] may be anywhere in the transfer buffer, often will actually be right after header	index(Datagram[1])	Repeat if/as needed
	len(Datagram[1])	
		Zero padding may be inserted here if convenient
	0	Required termination of header
	0	

L bytes

Figure 3-2 - NTB layout details (32 bit)

3.2   NCM Transfer Headers

3.2.1   NTH for 16-bit NTB (NTH16)

A 16-bit NTB shall begin with an NTH16 structure, described in Table 3-1.

Table 3-1: Sixteen Bit NCM Transfer Header (NTH16)

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x484D434E)	Signature of the NTH16 Header. This is transmitted in little-endian form, i.e., as 0x4E, 0x43, 0x4D, 0x48, or as the character sequence "NCMH"
4	wHeaderLength	2	Number (0x000C)	Size in bytes of this NTH16 structure, in little-endian format.
6	wSequence	2	Number	Sequence number. The transmitter of a block shall set this to zero in the first NTB transferred after every "function reset" event, and shall increment for every NTB subsequently transferred. The effect of an out-of-sequence block on the receiver is not specified. The specification allows the receiver to decide whether to check the sequence number, and to decide how to respond if it's incorrect. The sequence number is primarily supplied for debugging purposes.
8	wBlockLength	2	Number	Size of this NTB in bytes ("L" in Figure 3-1). Represented in little-endian form. NTB size (IN/OUT) shall not exceed *dwNtbInMaxSize* or *dwNtbOutMaxSize* respectively; see Table 7-3 in 7.2.1.
				If *wBlockLength* = 0x0000, the block is terminated by a short packet. In this case, the USB transfer must still be shorter than *dwNtbInMaxSize* or *dwNtbOutMaxSize*. If exactly *dwNtbInMaxSize* or *dwNtbOutMaxSize* bytes are sent, and the size is a multiple of *wMaxPacketSize* for the given pipe, then no ZLP shall be sent.
				wBlockLength= 0x0000 must be used with extreme care, because of the possibility that the host and device may get out of sync, and because of test issues.
				wBlockLength = 0x0000 allows the sender to reduce latency by starting to send a very large NTB, and then shortening it when the sender discovers that there's not sufficient data to justify sending a large NTB.
10	wNdpIndex	2	Number	Offset, in little endian, of the first NDP16 from byte zero of the NTB. This value shall be a multiple of 4, and must be >= 0x000C.

3.2.2 NTH for 32-bit NTB (NTH32)

The 32-bit form of the NTH is described in Table 3-2.

Table 3-2: Thirty-two bit NCM Transfer Header (NTH32)

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x686D636E)	Signature of the NTH32 Header. This is transmitted in little-endian form, i.e., as 0x6E, 0x63, 0x6D, 0x68, or as the character sequence "ncmh"
4	wHeaderLength	2	Number (0x0010)	Size in bytes of this NTH32 structure, in little-endian format.
6	wSequence	2	Number	Sequence number. The transmitter of a block shall set this to zero in the first NTB transferred after every "function reset" event, and shall increment for every NTB subsequently transferred. The effect of an out-of-sequence block on the receiver is not specified. The specification allows the receiver to decide whether to check the sequence number, and to decide how to respond if it's incorrect. The sequence number is primarily supplied for debugging purposes.
8	dwBlockLength	4	Number	Size of this NTB in bytes ("L" in Figure 3-2). Represented in little-endian form. NTB size (IN/OUT) shall not exceed *dwNtbInMaxSize* or *dwNtbOutMaxSize* respectively; see Table 7-3 in 7.2.1. If *dwBlockLength* = 0x0000, the block is terminated by a short packet. In this case, the USB transfer must still be shorter than *dwNtbInMaxSize* or *dwNtbOutMaxSize*. If exactly *dwNtbInMaxSize* or *dwNtbOutMaxSize* bytes are sent, and the size is a multiple of *wMaxPacketSize* for the given pipe, then no ZLP shall be sent. *dwBlockLength* = 0x0000 must be used with extreme care, because of the possibility that the host and device may get out of sync, and because of test issues. *dwBlockLength* = 0x0000 allows the sender to reduce latency by starting to send a very large NTB, and then shortening it when the sender discovers that there's not sufficient data to justify sending a large NTB.
12	dwNdpIndex	4	Number	Offset, in little endian, of the first NDP32 from byte zero of the NTB. This value must be a multiple of 4, and must be >= 0x0010 (because the first NDP32 had to be after the end of the NTH32).

3.3   NCM Datagram Pointers (NDPs)

NCM Datagram Pointers (NDPs) describe the Ethernet datagrams that are embedded in an NDP. As with NTH structures, two forms are defined. One form (the NDP16) is used for 16-bit NTBs; a second form (the NDP32) is used for 32-bit NTBs. These forms are architecturally equivalent, but differ in that many fields are 16-bits wide in the NDP16, but are 32-bits in the NDP32.

3.3.1   NDP for 16-bit NTBs (NDP16)

The layout of the NDP16 is given in Table 3-3. It has the following overall structure:

- 8 bytes of header information

- 1 or more NCM Datagram Pointer Entries (4 bytes per entry)

- A terminating zero NCM Datagram Pointer Entry (4 bytes).

Table 3-3: Sixteen-bit NCM Datagram Pointer Table (NDP16)

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x304D434E, 0x314D434E)	Signature of this NDP16. This is transmitted in little-endian form, i.e., as 0x4E, 0x43, 0x4D, 0x30 or 0x4E, 0x43, 0x4D, 0x31. or as the character sequences "NCM*0*",or "NCM1 where "0" or '1" has the meaning given in Table 3-5.
4	wLength	2	Number	Size of this NDP16, in little-endian format. This must be a multiple of 4, and must be at least 16 (0x0010).
6	wNextNdpIndex	2	Reserved (0)	Reserved for use as a link to the next NDP16 in the NTB
8	wDatagramIndex[0]	2	Number	Byte index, in little endian, of the first datagram described by this NDP16. The index is from byte zero of the NTB. This value must be >= the value stored in *wHeaderLength* of the NTH16 (because it must point past the NTH16).
10	wDatagramLength[0]	2	Number	Byte length, in little endian, of the first datagram described by this NDP16. For Ethernet frames, this value must be >= 14.
12	wDatagramIndex[1]	2	Number	Byte index, in little endian, of the second datagram described by this NDP16. If zero, then this marks the end of the sequence of datagrams in this NDP16.
14	wDatagramLength[1]	2	Number	Byte length, in little endian, of the second datagram described by this NDP16. If zero, then this marks the end of the sequence of datagrams in this NDP16.
			…	
wLength-4	wDatagramIndex[(wLength-8) / 4 - 1]	2	Number (0)	Always zero
wLength – 2	wDatagramLength[(wLength-8) / 4 - 1]	2	Number(0)	Always zero

3.3.2 NDP for 32-bit NTBs (NDP32)

The layout of the NDP32 is given in the Table 3-4. It has the following overall structure:

- 16 bytes of header information

- 1 or more NCM Datagram Pointer Entries (8 bytes per entry)

- A terminating zero NCM Datagram Pointer Entry (8 bytes).

Table 3-4: Thirty-two bit NCM Datagram Pointer Table (NDP32)

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x306D636E, 0x316D636E)	Signature of this NDP32. This is transmitted in little-endian form, i.e., as 0x6E, 0x63, 0x6D, 0x30 or 0x6E, 0x63, 0x6D, 0x31. These are equivalent to the character sequences "ncm0" and "ncm1", where "0" and "1" have the meaning given in Table 3-5.
4	wLength	2	Number	Size of this NDP32, in little-endian format. This must be a multiple of 8, and must be at least 32 (0x0020).
6	wReserved6	2	Reserved (0)	Reserved for future use.
8	dwNextNdpIndex	4	Number	Reserved for use as a link to the next NDP32 in the NTB
12	dwReserved12	4	Reserved (0)	Reserved for future use.
16	dwDatagramIndex[0]	4	Number	Byte index, in little endian, of the first datagram described by this NDP32. The index is from byte zero of the NTB. This value must be >= 0x0010 (because it must point past the NTH).
20	dwDatagramLength[0]	4	Number	Byte length, in little endian, of the first datagram described by this NDP32. For Ethernet frames, this value must be >= 14.
24	dwDatagramIndex[1]	4	Number	Byte index, in little endian, of the second datagram described by this NDP32. If zero, then this marks the end of the sequence of datagrams in this NDP32.
28	dwDatagramLength[1]	4	Number	Byte length, in little endian, of the second datagram described by this NDP32. If zero, then this marks the end of the sequence of datagrams in this NDP32.
	…			
wLength − 8	dwDatagramIndex[(wLength-8) / 8 - 1]	4	Number (0)	Always zero
wLength − 4	dwDatagramLength[(wLength-8) / 8 - 1]	4	Number(0)	Always zero

3.3.3   Datagram Formatting

All the datagrams described by a given NDP share a common format. The datagram always starts with a 14-byte [IEEE802.3] header, and then continues with the appropriate payload. The preparer of an NDP16 or NDP32 must choose whether a CRC-32 will be appended to the payload. If a CRC-32 is appended, and the header and payload combined are less than the minimum specified in 802.3, the datagram must be padded appropriately before calculating and appending the CRC-32.

Table 3-5: NDP Datagram Formatting Codes

Value	Datagram Formatting
0x30 ('0')	[IEEE802.3], no CRC-32
0x31 ('1')	[IEEE802.3], with CRC-32. If CRC-32 is appended, transmitter is responsible for padding the datagram to the appropriate minimum length for 802.3 prior to calculating and appending CRC-32.

3.3.4   NCM Ethernet Frame Alignment

It is well known that many network stacks in embedded devices benefit through careful alignment of the payload to system defined memory boundaries. NCM allows a function to align transmitted datagrams on any convenient boundary within the NTB. Functions indicate how they intend to align their transmitted datagrams to the host in the NTB Parameter Structure (Table 6-3).

Similarly, for data transmitted from the host, functions indicate their preferred alignment requirements to the host. The host then formats the NTBs to satisfy this constraint.

NCM assumes that hosts are more flexible and powerful than devices. Therefore, the host shall always honor the constraints given by the device when preparing OUT NTBs.

Alignment requirements are met by controlling the location of the payload (the data following the Ethernet header in each datagram). This alignment is specified by indicating a constraint as a divisor and a remainder. The agent formatting a given NTB aligns the payload of each datagram by inserting padding, such that the offset of each datagram satisfies the constraint:

$$\text{Offset} \% \ wNdpInDivisor == wNdpInPayloadRemainder \ \text{(for IN datagrams)}$$

Or

$$\text{Offset} \% \ wNdpOutDivisor == wNdpOutPayloadRemainder \ \text{(for OUT datagrams)}$$

Two use cases are anticipated (although the specification doesn't restrict the user to these two cases).

In one use case, the function wants to align the beginning of an IP packet to a cache line boundary. Cache lines are generally much smaller than the maximum Ethernet frame size. In this case, the *wNdpXxDivisor* is set to the size of a cache line (in bytes), and the *wNdpXxPayloadRemainder* is set to zero. The effect is shown schematically in Figure 3-3.

In another use case, the function wants to place each IP packet in a fixed sized buffer. (This is primarily intended for use on the OUT pipe.) For this to work, each buffer must be bigger than the maximum Ethernet datagram size[1]. In this case, *wNdpXxDivisor* is set to the size of the buffer, and *wNdpXxPayloadRemainder* is set to the desired offset of the IP packet in the fixed size buffer. This situation is shown schematically in Figure 3-4.
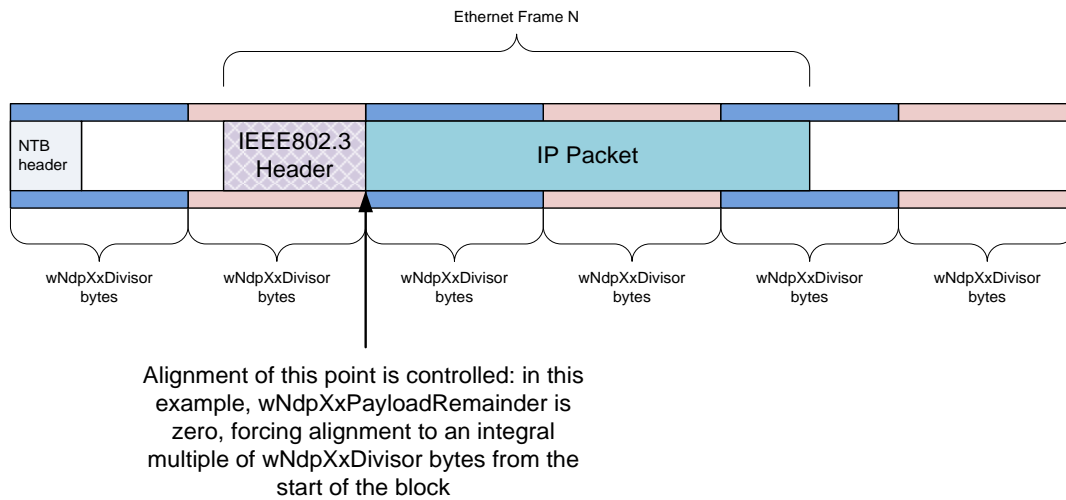
Figure 3-3 - Alignment to a cache line
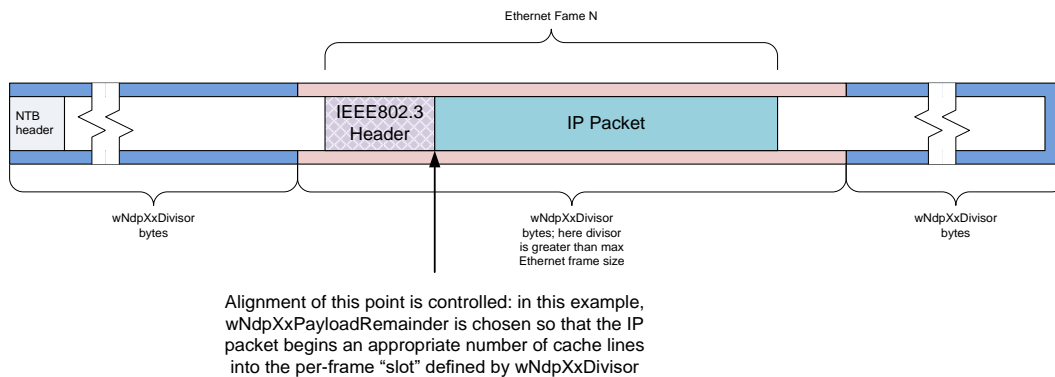
Figure 3-4 - Alignment for fixed-size internal buffers

3.4   NTB Maximum Sizes

NCM functions determine the maximum permitted size of NTB that they can process. The upper limit is usually determined by the buffering capacity of the attached function, but there may be other factors involved as well, such as latency.

---

[1] Refers to the maximum datagram size in host memory. For NCM 1.1 functions that support Large Send Offload, this may be much larger than the maximum Ethernet frame length on the wire. Refer to sections 4.7 and 8.3.4 for additional details.

For OUT pipes, the host determines the actual size of the NTB data structures sent to the device. Hosts can discover the maximum size supported by the device from the NTB Parameter Structure (see 7.2.1), and shall not send NTBs larger than the device can support.

For IN pipes, the host tells the device the size of NTB data structures that it wishes the device to send using the *SetNtbInputSize* command. Hosts can discover the maximum size supported by the device from the field *dwNtbInMaxSize* in the NTB Parameter Structure (Table 7-3). Devices shall not send NTBs larger than the host has requested. The host shall not select a maximum NTB size that is not supported by the device.

3.5   NTB format support

Functions conforming to this specification shall support 16-bit NTB structures (Table 3-1 and Table 3-3). Functions may also support 32-bit NTB structures (Table 3-2 and Table 3-4).

3.6   Ethernet frame Datagram Maximum Size

The maximum size of an Ethernet frame datagram can be dynamically adjusted by the host using the *SetMaxDatagramSize* request. Commonly, Ethernet frames are 1514 bytes or less in length (not including the CRC), but for many applications a larger maximum frame size is needed (e.g., 802.1Q VLAN tagging, jumbo frames). Hosts can discover the maximum Ethernet frame size supported by a device from the value *wMaxSegmentSize* in the Ethernet Networking Functional Descriptor (see Table 6-1 and [USBECM12], 5.4), and shall not select a size larger than the device can support, nor shall it send a frame larger than the device can support.

Host or function may append CRCs to datagrams. These four-byte CRCs are *not* included when determining the "maximum segment size", but they are counted when specifying the datagram size in the NDP. For example, if the maximum datagram size is currently 1514, and the NDP header indicates that CRCs are being appended, then the maximum *wDatagramLength* value for any datagram in that NDP is 1518.

3.7   Null NCM Datagram Pointer Entries

Any NCM Datagram pointer entry with an index field (*wDatagramIndex*, *dwDatagramIndex*) of zero or with a length field (*wDatagramLength*, *dwDatagramLength*) of zero, or with both index and length fields set to zero, shall be treated as a Null entry. Receivers shall process datagram pointer entries sequentially from the first entry in the NTB. The first Null entry shall be interpreted as meaning that all following NCM Datagram Pointer Entries in the NDP are to be ignored.

The rules given in sections 3.3.1 and 3.3.2 specify that every NDP shall end with a Null entry. It is an error for a transmitter to format an NDP without a terminating Null entry. Receivers MAY discard such NDPs (and all associated frames) entirely.

Transmitters are allowed to send a properly-formatted NTB containing an NDP whose datagram pointer entries are all zero. Receivers shall ignore such NTBs.

It is an error for a transmitter to send an NDP with non-Null NCM Datagram Pointer Entries following the first Null. Receivers MAY process datagrams up to the first Null NCM Datagram Pointer Entry, and MAY ignore the remaining non-Null entries in the NDP.

4   NCM 1.1 Data Transport

4.1   Overview

The data transport of NCM 1.1 builds upon the core components of the data transport of NCM 1.0 (section 3), but rather than being a superset, it contains several key differences. It offers extended capabilities on the one hand, yet is more restrictive on the other hand. The purpose of these restrictions is to simplify at once NCM function and device driver implementations.

NCM 1.1 carries over the concepts of NCM Transfer Block (NTB), NCM Transfer Header (NTH) and NCM Datagram Pointer (NDP), with several modifications. The NTB for NCM 1.1. is known as "Extended NTB" or NTBX. A schematic diagram is shown in Figure 4-1.

- Only 32-bit NTBXs are supported, so there is only one type of header (NTHX) and one type of datagram pointer (NDPX). This has the following consequences:

  - The bmNtbFormatsSupported field of *GetNtbParameters* (7.2.1) is meaningless and should be ignored by a host operating in NCM 1.1 mode.
  - The *SetNtbFormat* (7.2.4) and *GetNtbFormat* (7.2.5) commands, are unused and shall be ignored by a function operating in NCM 1.1 mode.

- Each NDPX points to a single Ethernet frame encapsulated within the NTBX. An exception to this is the last NDPX in the NTBX, which optionally may not point to any frame. A schematic diagram of such an NTBX is shown in Figure 4-2.

- Each NDPX points to the next NDPX, except the last NDPX of the NTBX.

- The NTBX is arranged so that each Ethernet frames always follows the NDPX that points to it, and always precedes the following NDPX (if such exists). However, they are not required to be contiguous in memory (the NTB may contain padding).
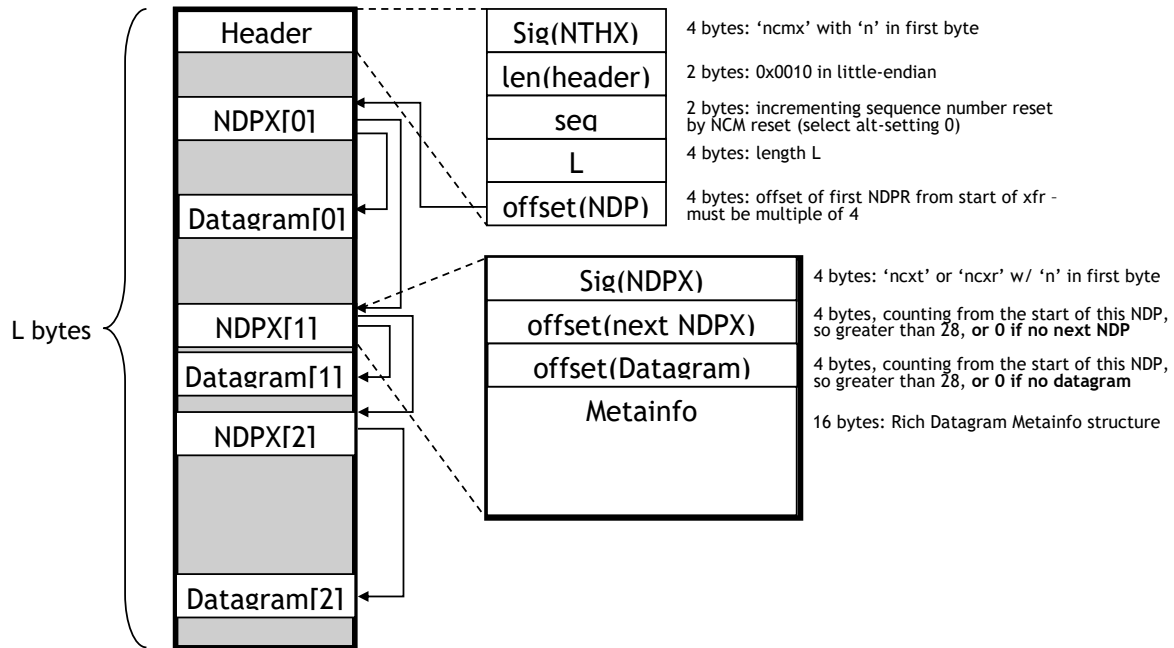
Figure 4-1 – Extended NTB layout details
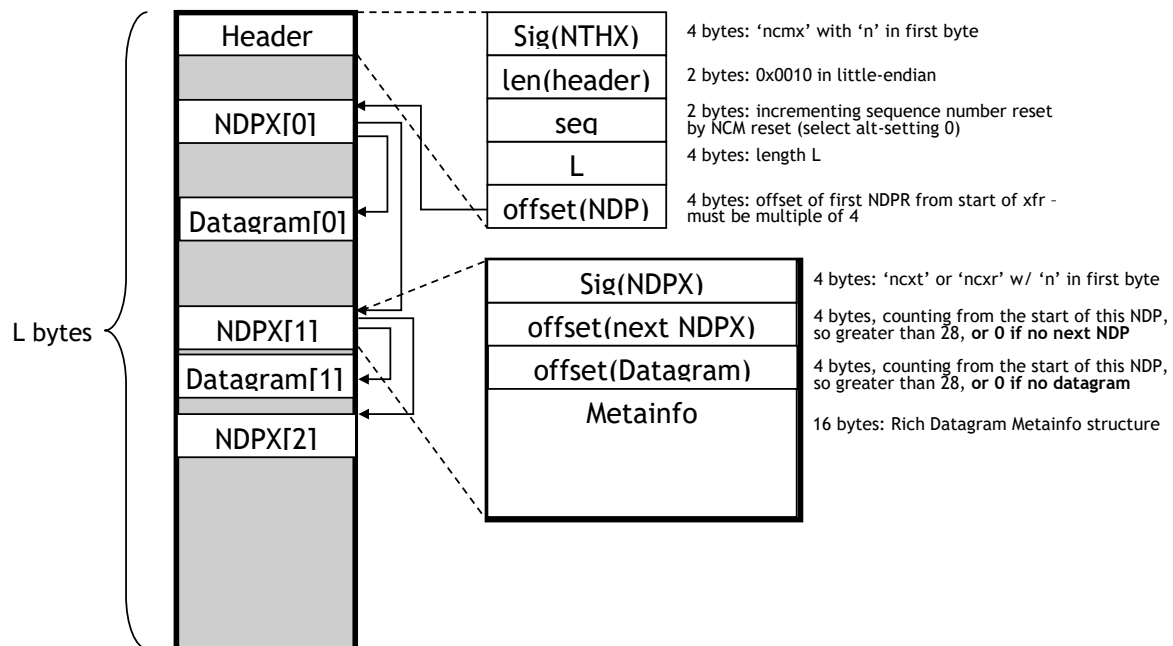
Figure 4-2 – Extended NTB layout with no datagram in last NDPX

4.2   Extended NCM Transfer Header

The NCM 1.1 Extended NTH is described in Table 4-1.

Table 4-1: Extended NCM Transfer Header (NTHX) for NCM 1.1

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x786D636E)	Signature of the NTHX Header. This is transmitted in little-endian form, i.e., as 0x6E, 0x63, 0x6D, 0x78, or as the character sequence "ncmx".
4	wHeaderLength	2	Number (0x0010)	Size in bytes of this NTHX structure, in little-endian format.
6	wSequence	2	Number	Sequence number. The transmitter of a block shall set this to zero in the first NTB transferred after every "function reset" event, and shall increment for every NTB subsequently transferred. The effect of an out-of-sequence block on the receiver is not specified. The specification allows the receiver to decide whether to check the sequence number, and to decide how to respond if it's incorrect. The sequence number is primarily supplied for debugging purposes.
8	dwBlockLength	4	Number	Size of this NTB in bytes ("L" in Figure 4-1). Represented in little-endian form. NTB size (IN/OUT) shall not exceed *dwNtbInMaxSize* or *dwNtbOutMaxSize* respectively; see Table 7-3 in 7.2.1. If *dwBlockLength* = 0x0000, the block is terminated by a short packet. In this case, the USB transfer must still be shorter than *dwNtbInMaxSize* or *dwNtbOutMaxSize*. If exactly *dwNtbInMaxSize* or *dwNtbOutMaxSize* bytes are sent, and the size is a multiple of *wMaxPacketSize* for the given pipe, then no ZLP shall be sent. *dwBlockLength* = 0x0000 allows the sender to reduce latency by starting to send a very large NTB, and then shortening it when the sender discovers that there's not sufficient data to justify sending a large NTB.
12	dwNdpIndex	4	Number	Offset, in little endian, of the first NDPX from byte zero of the NTB. This value must be a multiple of 4, and must be >= 16, because the offset is counted from byte 0 of the NTHX.

4.3   Extended NCM Datagram Pointer (NDPX)

An Extended NCM Datagram Pointer (NDPX), used in NCM 1.1 data transport, describes a single Ethernet datagram, including its payload, and any accompanying meta information used in the transmission or reception of the payload. The high-level structure of transmit and receive NDPXs is similar, however the contents of the meta info structure is different. For this purpose, the transmit/receive NDPXs use different signatures. This allows a debugger / memory analyzer to figure out quickly whether a particular NDPX describes a datagram sent from the host to the device (transmit direction) or from the device to the host (receive direction).

The layout of the NDPX is given in the Table 4-2. Each NDPX describes a single datagram, and contains a pointer to the next NDPX. One the exception is the last NDPX in the NTB, which does not point to a next NDPX and may, <u>optionally,</u> not point to a datagram. An NDPX not pointing to a datagram is known as "null NDPX".

Table 4-2: Extended NCM Datagram Pointer (NDPX)

Offset	Field	Size	Value	Description
0	dwSignature	4	Number (0x7478636E or 0x7278636E)	Signature of this Extended NDP. This is transmitted in little-endian form.
				For a transmit NDPX (host to device): 0x6E, 0x63, 0x78, 0x74 or as the character sequence "ncxt".
				For a receive NDPX (device to host): 0x6E, 0x63, 0x78, 0x72 or as the character sequence "ncxr".
4	dwNextNdpOffset	4	Number	Byte Offset, in little endian, of the next Extended NDP. This value must be a multiple of 4, and >= 28, because it is counted from byte 0 of this NDPX. **A value of zero denotes that this is the last Extended NDP within an NTB.**
8	dwDatagramOffset	4	Number	Byte Offset, in little endian, of the datagram described by this Extended NDP. The offset is from byte zero of this Extended NDP, and so must be greater than 28. **A value of zero denotes that this Extended NDP does not contain a datagram and shall only be used for the last Extended NDP in an NTB. If the value of this field is zero, then** *dwNextNdpOffset* **shall also be zero.**
12	stMetaInfo	16	Structure	Extended Datagram Meta-info structure. This field shall contain meta-information for the datagram transmitted by this Extended NDP.
				The content of the meta-info for a transmit NDPX is described in Section 4.4.1.
				The content of the meta-info for a receive NDPX is described in Section 4.4.3.

4.4 Meta-info structures for Extended NDP

The meta-info structure describes the datagram associated with the NDPX. Some of the meta-info is relevant in either transmit or receive directions, while other is relevant for both, therefore separate structures are defined for transmit and receive.

To simplify NDPX processing, for the NCM function and the host alike, an attempt was made to place fields used by both traffic directions in the same offsets within the structures. As a consequence, fields that are only relevant to one of the structures may be marked as 'reserved' in the other. Additionally, extra 'reserved' bytes have been allocated for purposes of alignment and to support future expansions of the standard.

4.4.1 Transmit meta-info structure

The transmit meta-info structure is shown in Table 4-3.

Table 4-3: Meta-info structure for transmit NDPX

	Transmit Datagram Meta-info																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Flags								Reserved				Length																			
1	MSS																TxRequest															
2	Reserved				L3Length								VLAN																			
3	Reserved																L2Length								L4Length							

DWORD	Bits	Field	Value	Description
0	[19:0]	Length	Number	**Total length of the datagram to which this Meta-info applies** Current implementations are not expected to use more than 17 bits, due to the packet length limits of TCP/IP stacks.
0	[23:20]	Reserved	0	Should be set to 0.
0	[31:24]	Flags	Bit field	**Packet Flags (8 bits)** 1:0 – L4 type: 00 = Unknown / No L4,01 = Reserved, 10 = TCP, 11 = UDP 3:2 – L3 type: 00 = Unknown / No L3, 01 = Reserved, 10 = IPv4, 11 = IPv6 7:4 – Reserved
1	[15:0]	TxRequest	Bit field	**Transmit Request Flags (16 bits)** **A value of 1 in the following fields denotes the Host request for Device to perform the requested offload/feature on this datagram** 0 – Append L2 FCS (MAC CRC) 1 – Insert L3 Header Checksum (IPv4 Checksum) 2 – Insert L4 Checksum (TCP/UDP Checksum) 3 – Insert VLAN Tag 4 – Perform Large Send Offload (TCP/UDP Segmentation) 15:5 – Reserved
1	[31:16]	MSS	Number	**Maximum Segment Size** The maximum size of the payload (excluding headers) for this datagram. Applicable only when Large Send Offload is enabled. This should be set so that the total length of each datagram (MSS + L2Length + L3Length + L4Length) does not exceed the maximum permitted frame size in the network. See section 4.7 for additional discussion.

DWORD	Bits	Field	Value	Description
2	[15:0]	VLAN	Structure	**IEEE 802.1Q VLAN tag format (16 bits)** **The value of the VLAN tag to insert into the datagram** As defined in [IEEE802.1Q] 11:0 – 12-bit VLAN ID 12 – DEI (Drop Eligible Indicator) 15:13 – PCP (Priority Code Point)
2	[24:16]	L3Length	Number	**Layer 3 Header Length (9 bits)** Applicable when Large Send Offload or L4/L3 Checksum are enabled. See Figure 4-3.
2	[31:25]	Reserved	0	Should be set to 0.
3	[7:0]	L4Length	Number	**Layer 4 Header Length (8 bits)** Applicable when Large Send Offload or L4 Checksum are enabled. See Figure 4-3.
3	[14:8]	L2Length	Number	**Layer 2 Header Length (7 bits)** Applicable when Large Send Offload or L4/L3 Checksum are enabled. See Figure 4-3.
3	[31:15]	Reserved	0	Should be set to 0.

Figure 4-3 shows the structure of a datagram and the values of related meta-info fields, for the typical case of *TxRequest[0]=1*. Refer Table 4-4 and Table 4-5 for additional details.
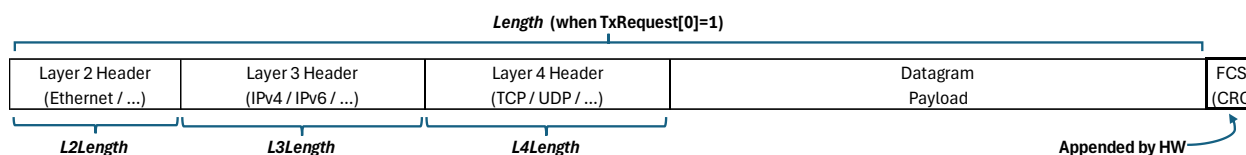
Figure 4-3 – Network datagram structure and values of related meta-info fields

4.4.2 Transmit parameters combinations and dependencies

The following tables specify the dependencies of the various Transmit parameters:

Table 4-4: Dependencies between TxRequest Insert L2 FCS and other fields

L2 FCS <u>Disabled</u> (TxRequest[0]=0)	The datagram shall include the correct FCS (CRC) value as computed by the software. *Length* refers to the length of the datagram, including the FCS. All other offloads in *TxRequest* <u>shall</u> be disabled. *VLAN*, *MSS* and *L2/L3/L4 Length* fields are meaningless (should be set to 0). *Flags* field is meaningless.
L2 FCS <u>Enabled</u> (TxRequest[0]=1)	The datagram shall not include the FCS. *Length* refers to the length of the datagram without the 4 FCS (CRC) bytes that shall be appended by the NCM function. Other offloads may be set as described below.

Table 4-5: Dependencies between different offloads in TxRequest

L3 Checksum <u>Enabled</u> (TxRequest[1]=1)	*L2Length*, *L3Length* and *Flags.L3Type* shall be set according to header lengths and types. Note: IPv6 header has no checksum field, so setting *L3Type* to IPv6 will lead to checksum not being computed, even if requested. Other offloads may be enabled as described below.
L4 Checksum <u>Enabled</u> (TxRequest[2]=1)	*L2/L3/L4 Length* and *Flags.L3Type/L4Type* fields shall be set according to header lengths and types. Other offloads may be enabled as described below.
VLAN <u>Enabled</u> (TxRequest[3]=1)	The NCM function shall add 4 bytes to the packet – a 2-byte VLAN Ethertype (0x8100), and a 2-byte tag added taken from the *VLAN* field. The values of *Length* and *L2Length* shall <u>not</u> include the additional 4 bytes.
LSO <u>Enabled</u> (TxRequest[4]=1)	*Length* shall specify the full length of the <u>pre-segmentation</u> datagram, including all headers, but <u>excluding</u> CRC and the additional 4 bytes of the VLAN offload (if enabled). *Flags* shall specify the correct L3/L4 protocols. *L4 Checksum* shall be enabled, and *L3 Checksum* shall be enabled if the packet is IPv4. *L2/L3/L4Length* shall specify the actual lengths of the respective headers. *MSS* shall specify the desired payload size (<u>excluding headers</u>) of each segment packet, and should be computed by the software according to the known values of the maximum datagram size (see 7.2.8, 7.2.9) and of the header lengths.

In the typical case, values for the *L2Length*, *L3Length* and *L4Length* fields are provided by the software layer that generates the data for the respective headers. Specific implementations may ignore these fields and perform packet parsing internally to arrive at the correct values, but, in general, the behavior of an NCM function is undefined if these values are not set correctly.

4.4.3 Receive meta-info structure

The receive meta-info structure is shown in Table 4-6.

Table 4-6: Meta-info structure for receive NDPX

	Receive Datagram Meta-info																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Flags								RSSType				Length																			
1	RxErrors																RxStatus															
2	Reserved																VLAN															
3	RSSHash																															

DWORD	Bits	Field	Value	Description
0	[19:0]	Length	Number	**Total length of the datagram to which this Meta-info applies** Current implementations are not expected to use more than 17 bits, due to the packet length limits of TCP/IP stacks.
0	[23:20]	RSSType	Value	**RSS hash type that was calculated on the packet** See section 10.2 for additional details. 0 – No hash computation / RSS disabled 1 – IPv4 2 – TCP/IPv4 3 – IPv6 4 – IPv6 with Extensions 5 – TCP/IPv6 6 – TCP/IPv6 with Extensions 7 – UDP/IPv4 8 – UDP/IPv6 9 – UDP/IPv6 with Extensions 10-15 - Reserved
0	[31:24]	Flags	Bit field	**Packet Flags (8 bits)** 1:0 – L4 type: 00 = Unknown / No L4, 01 = Reserved, 10 = TCP, 11 = UDP 3:2 – L3 type: 00 = Unknown / No L3, 01 = Reserved, 10 = IPv4, 11 = IPv6 7:4 – Reserved
1	[15:0]	RxStatus	Bit field	**Receive Status Flags (16 bits)** **A value of 1 in the following fields denotes that the device performed the specific offload/feature on this datagram** 0 – L2 FCS (MAC CRC) stripping 1 – L3 Header Checksum (IPv4 Checksum) verification 2 – L4 Checksum (TCP/UDP Checksum) verification 3 – VLAN Tag stripping 4 – Large Receive Offload (Receive coalescing) 15:5 – Reserved
1	[31:16]	RxErrors	Bit field	**Receive Error Flags (16 bits)** **A value of 1 in the following fields denotes that the device detected an error while performing the specific offload/feature** 0 – L2 FCS (MAC CRC) error 1 – L3 Header Checksum (IPv4 Checksum) error 2 – L4 Checksum (TCP/UDP Checksum) error 15:3 – Reserved

DWORD	Bits	Field	Value	Description
2	[15:0]	VLAN	Structure	**IEEE 802.1Q VLAN tag format (16 bits)** **The value of the VLAN Tag stripped from this datagram** As defined in [IEEE802.1Q] 11:0 – 12-bit VLAN ID 12 – DEI (Drop Eligible Indicator) 15:13 – PCP (Priority Code Point)
2	[31:16]	Reserved	0	Should be set to 0.
3	[31:0]	RSSHash	Number	**32-bit RSS Hash Value as computed over the packet** This value is valid only if *RSSType* is not zero and not one of the reserved values. The fields over which the RSS hash is computed are described in Table 10-1.

4.4.4 Receive parameters relations and dependencies

The following tables specify the dependencies of the various Receive parameters:

Table 4-7: Dependencies between RSS Hash and RSS Type fields

RSSType=0,10..15	*RSSHash* field is meaningless. NCM function is expected to set *RSSHash* to 0.
RSSType=1..9	*RSSHash* field contains the 32-bit Toeplitz hash computed over the appropriate fields of the datagram, according to the value of *RSSType*. Refer to Table 10-1 in 10.2 for additional details.

Table 4-8: Dependencies between FCS bits in RxStatus and RxErrors

RxStatus[0]	*RxErrors[0]*	
0	0	FCS (CRC) is present in the datagram and is valid. *Length* field includes the 4-byte FCS.
0	1	FCS (CRC) is present in the datagram and is invalid. *Length* field includes the 4-byte FCS.
1	0	FCS (CRC) was valid and was stripped from the datagram. *Length* field does not include the FCS.
1	1	FCS (CRC) was invalid and was from the datagram. *Length* field does not include the FCS.

Table 4-9: Dependencies between RxStatus and other fields

RxStatus[0]=1 **(L2 FCS Strip)**	See Table 4-8.
RxStatus[1]=1 **(L3 Checksum)**	Packet was detected as IPv4 and checksum was verified. *Flags* should be set according to the L4 header type present in the packet. If the IPv4 checksum was invalid, *RxErrors[1]* shall be set.
RxStatus[2]=1 **(L4 Checksum)**	Layer 4 (TCP/UDP) checksum was verified. *Flags* should be set according to the L3 and L4 header types present in the packet. If the Layer 4 checksum was invalid, *RxErrors[2]* shall be set.
RxStatus[3]=1 **(VLAN Strip)**	VLAN was detected in the packet, and the 4 VLAN bytes were stripped– a 2-byte VLAN Ethertype (0x8100), and a 2-byte tag. The VLAN tag shall be placed in the *VLAN* field of the meta-info structure. The *Length* field shall <u>not</u> include the 4 bytes that were stripped.
RxStatus[4]=1 **(Large Receive Offload)**	The *Length* field indicates the total length of the <u>reassembled</u> datagram, including all headers. *RxErrors* field shall <u>not</u> have any bit set. *RxStatus[0]* shall be set, as the L2 FCS is not computed for the reassembled datagram, and the 4 bytes FCS shall not be present in the datagram.

4.5   Layer 2 Frame Check Sequence (CRC)

Per the Ethernet specification [IEEE802.3], any transmitted frame shall have a valid 4-byte frame check sequence (CRC) appended to it. During nominal operation, frames without a valid CRC shall be rejected by the receiver.

On the transmit side, the Host software may compute the CRC and include it in the datagram buffer within the NTBX, and in this case it shall **clear** the bit *stMetaInfo.TxRequest[0]*. Alternatively, the software may supply the datagram without the CRC, and request that the Device computes and appends it before transmitting the frame on the wire; in this case, the software shall **set** the bit *stMetaInfo.TxRequest[0]*.

On the receive side, in nominal operation, the Device strips the CRC from packets that pass CRC verification before forwarding them to the Host. Packets that fail CRC verification are discarded by the NCM function and are not passed to the Host. This behavior can be modified via the Receive Offloads configuration (section 8.3.5). Software can deduce the validity of the checked CRC and its presence in the datagram via the values of *stMetaInfo.RxStatus[0]* and *stMetaInfo.RxErrors[0]*. Refer to Table 4-8 for additional details.

4.6   NTB Maximum Sizes

The NTB size considerations described in section 3.4 apply to NCM 1.1 as well.

4.7   Ethernet frame Datagram Maximum Size

The maximum size of an Ethernet frame datagram can be dynamically adjusted by the host using the *Set-MaxDatagramSize* request. Commonly, Ethernet frames include 1500 bytes of payload, or less (the 14 bytes of the header, 802.1Q VLAN tag, and CRC are not counted towards the limit). Some network interface cards, as well as some switches, running at speeds of 1 Gigabit per second, or higher, may support larger sizes, such as 4000 bytes, 9000 bytes, or more. While the Ethernet standard mandates support for 1500-byte frames, support for higher sizes (also known as "jumbo frames") is not required, and compatibility is not guaranteed. Hosts can discover the maximum Ethernet frame size supported by a device from the value *wMaxSegmentSize* in the Ethernet Networking Functional Descriptor ([USBECM12], 5.4).

NCM 1.1 functions may support Large Send and Large Receive offloads for standard protocols, such as TCP and UDP. With such functions, the "datagram buffer length" field of the *stMetaInfo* structure may be larger than *wMaxSegmentSize*.

When preparing the NTBs for transmission, it is the responsibility of the Host software to set the MSS field of the transmit *stMetaInfo* structure so that the size of any segmented frame on the wire shall not exceed *wMaxSegmentSize.* See section 4.4 for additional information.

4.8   Considerations for store-and-forward versus on-the-fly processing

The fundamental data unit of NCM 1.1 is the NTBX (see section 4.1). On the transmit path, data passes from the host, via the USB bus, to the NCM function, which extracts the datagrams from the NTBX, and sends them out to the network (e.g., on an Ethernet cable). On the receive path, data received from the network is processed by the NCM function, which builds NTBXs and passes them via the USB bus towards the host.

In the transmit direction, the NCM function may store (buffer) the entire NTBX, and in the receive direction it may buffer incoming datagrams until enough data has accumulated for a full NTBX. This is referred to as 'store-and-forward' approach.

Alternatively, the NCM function may begin transmitting frames on the wire before the NTBX is received in its entirety (transmit direction), or sending a partially constructed NTBX up towards the host, before the final size and contents are known (receive direction). This is known as 'on-the-fly processing'.

The choice of implementation depends, among other things, on memory capacity, speed and latency requirements. The example in Figure 4-4 compares the two approaches (in the receive direction). On-the-fly processing generally requires smaller on-chip buffers, as less data must be stored at the same time, and can support lower latencies. The trade-off is more complicated processing logic, which has to make decisions based on partial data. Although the exact details are beyond the scope of this specification, the definitions of the NDPX structures (sections 4.1, 4.2 and 4.3), and specifically, the guarantees on the relative ordering of NDPXs and datagrams, were selected so as not to preclude any possible implementation.
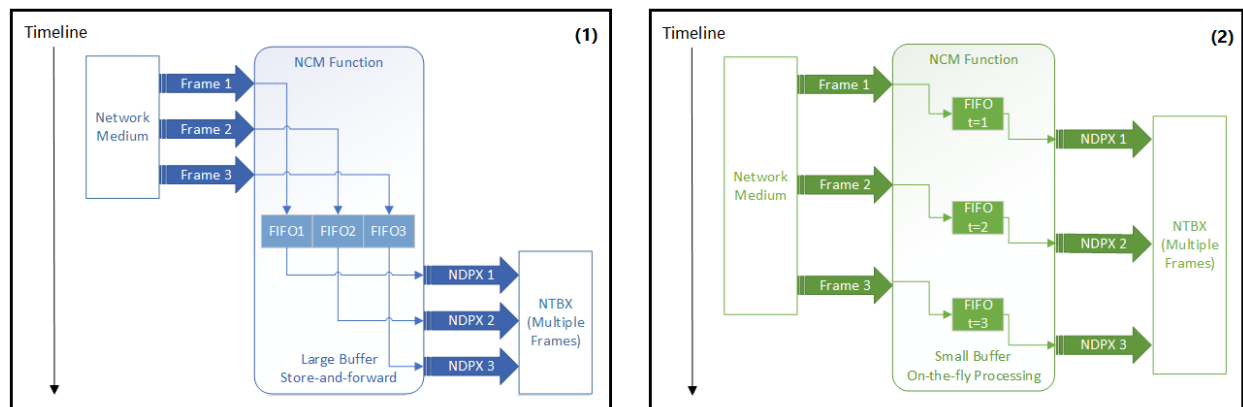
Figure 4-4 – Store-and-forward (1) versus on-the-fly processing (2)

5 Class-Specific Codes

This section lists the codes for the Communications Class and Data Class, specifically subclasses and protocols. These values are used in the *bInterfaceSubClass* and *bInterfaceProtocol* fields of the standard device descriptors as defined in [USBCORE]. Values for *bDeviceClass*, *bDeviceSubClass*, and *bDeviceProtocol* in the Device Descriptor are defined in [USBCDC12].

5.1 NCM Communications Interface Subclass Code

Table 5-1 defines the interface subclass code used in the NCM Communications Interface descriptor.

Table 5-1 NCM Communications Interface Subclass Code

Code	Subclass
0Dh	Network Control Model

5.2 NCM Communications Interface Protocol Code

Table 5-2 lists the Protocol code used in the NCM Communications Interface Descriptor. If the optional *SendEncapsulatedCommand* and *GetEncapsulatedResponse* commands are supported (as stated by the value in bit D2 of *bmNetworkCapabilities* in the NCM Functional Descriptor, Table 6-2), this code indicates the format of commands sent, and responses received via those commands. If *SendEncapsulatedCommand* and *GetEncapsulatedResponse* are not supported, then *bInterfaceProcotol* shall be set to zero.

Regardless of the format of encapsulated commands and responses, all NCM functions shall implement the default pipes requests and notifications as specified by Table 7-1 and Table 7-3.

Table 5-2 NCM Communications Interface Protocol Code

Code	Protocol	Defined By	Encapsulated Command/Response Payload Format
00h	None	[USBCDC12]	No encapsulated commands / responses.
01h	N/A	[USBCDC12]	Do not use.
02h	N/A	[USBCDC12]	Do not use
03h	N/A	[USBCDC12]	Do not use
04h	N/A	[USBCDC12]	Do not use
05h	N/A	[USBCDC12]	Do not use
06H	N/A	[USBCDC12]	Do not use
06h-FDh	Reserved		Reserved for future use
FEh	OEM defined	[USBWMC11]	External Protocol: Commands defined by Command Set Functional Descriptor following the NCM Communications Interface Descriptor.
FFh	N/A	[USBCDC12]	Do not use

5.3   NCM Data Class Interface Protocol Codes

[USBCDC12] defines Data Class Protocols. Alternate settings 0 and 1 of an NCM function's Data Interface shall use the protocol code defined in Table 5-3.

Table 5-3 NCM Data Class Protocol Code

Code	Protocol
01h	Network Transfer Block (3.3.3)

5.4   NCM Functional Descriptor Codes

Table 5-4: NCM Functional Descriptor Code

Code	Name	Descriptor
1Ah	NCM_FUNC_DESC_CODE	NCM Functional Descriptor
1Dh	NCM_EXT_CAPABILITY_DESC_CODE	NCM Extended Capability Descriptor
1Eh	NCM_EXT_FEATURE_DESC_CODE	NCM Extended Feature Descriptor

6   Descriptors

6.1   Standard USB Descriptor Definitions

Refer to [USBCDC12].

6.2   NCM Communications Interface Descriptor Requirements

NCM functions must implement class-specific descriptors ("functional descriptors") for the NCM Communications Interface. The framework for these is defined in [USBCDC12].

NCM Communications Interfaces must implement the following functional descriptors described in [USBCDC12]:

- Header Functional Descriptor (describing the level of compliance to [USBCDC12])

- Union Functional Descriptor (containing the interface numbers of the Communications Interface and the Data interface).

 NCM functions must implement an Ethernet Network Functional Descriptor, as defined in [USBECM12]

NCM functions must implement a NCM Functional Descriptor, as described in section 6.2.1.

If *bInterfaceProtocol* is FEh, then the NCM function must provide a Command Set Functional Descriptor, and may provide additional Command Set Detail Functional Descriptors.

The class-specific descriptors must be followed by an Interrupt IN endpoint descriptor.

Descriptor requirements are summarized in Table 6-1.

Table 6-1: NCM Communication Interface Descriptor Requirements

Descriptor	Description	Req'd/Opt	Order	Reference
HEADER	CDC Header functional descriptor	Required	First	[USBCDC12], 5.2.3.1
UNION	CDC Union functional descriptor	Required	Arbitrary	[USBCDC12] 5.2.3.2
ETHERNET	CDC Ethernet Networking Functional Descriptor	Required	Arbitrary	[USBECM12], 5.4
NCM	NCM Functional Descriptor	Required	Arbitrary	5.2.1
COMMAND SET	Command Set Functional Descriptor	Required if NCM Communications Interface bInterfaceProtocol is 0FEh	Arbitrary	6.2.2
COMMAND SET DETAIL	Command Set Detail Functional Descriptor	Optional if Command Set Functional Descriptor is present; otherwise prohibited	After Command Set Functional Descriptor	6.2.3

6.2.1   NCM Functional Descriptor

This descriptor provides information about the implementation of the NCM function. It is mandatory, and must appear after the Header Functional Descriptor.

Table 6-2: NCM Functional Descriptor

Offset	Field	Size	Value	Description
0	bFunctionLength	1	6	Size of Descriptor in bytes
1	bDescriptorType	1	Constant	CS_INTERFACE (0x24)
2	bDescriptorSubtype	1	Constant (1Ah)	NCM Functional Descriptor subtype, as defined in Table 5-4.
3	bcdNcmVersion	2	Number 0x0100	Release number of this specification in BCD, with implied decimal point between bits 7 and 8. 0x0100 == 1.00 == 1.0. This is a little-endian constant, so the bytes will be 0x00, 0x01. The value indicates that the device supports NCM 1.0 spec for legacy drivers.
5	bmNetworkCapabilities	1	Bitmap	Specifies the capabilities of this function. A bit value of zero indicates that the capability is not supported. D7: Function supports extended NCM capabilities (NCM 1.1). D6: Reserved (zero) D5: Function can process 8-byte forms of *GetNtbInputSize* and *SetNtbInputSize* requests D4: Function can process *SetCrcMode* and *GetCrcMode* requests D3: Function can process *SetMaxDatagramSize* and *GetMaxDatagramSize* requests. D2: Function can process *SendEncapsulatedCommand* and *GetEncapsulatedResponse* requests. D1: Function can process *GetNetAddress* and *SetNetAddress* requests. D0: Function can process *SetEthernetPacketFilter* requests, as defined in [USBECM12]. If not set, broadcast, directed and multicast packets are always passed to the host.

6.2.2   Command Set Functional Descriptor

If the NCM Communications Interface has *bInterfaceProtocol* set to "External Protocol", as given in Table 5-2, then the command set transported by *SendEncapsulatedCommand* and *GetEncapsulatedResponse* is governed by a specification external to this document. The specification is identified by a GUID given in a Command Set Functional descriptor, which must appear associated with the NCM Communications Interface descriptor. This descriptor is defined in [USBWMC11], section 8.1.2.2. The GUID is defined by the appropriate external specification. The GUID identifies the format and contents of the command set. The command set may be, but is not required to be, AT commands and responses.   This descriptor is required if *bInterfaceProtocol* is set to "External Protocol".

If the NCM Communications Interface has *bInterfaceProtocol* set to any other value, then the Command Set Functional Descriptor shall not appear, and the host shall ignore any such descriptors.

6.2.3 Command Set Detail Functional Descriptor

If a Command Set Functional Descriptor appears, it may be followed by one or more Command Set Detail Functional Descriptors, as described in [USBWMC11], section 8.1.2.3. If the Command Set Functional Descriptor is not present, Command Set Detail Functional Descriptors shall not appear, and the host shall ignore any such descriptors.

6.3 Data Interface Descriptor Requirements

The Data Interface of an NCM networking function shall have two alternate settings. The first alternate setting (the default interface setting, alternate setting 0) shall include no endpoints and therefore no networking traffic can be exchanged when the default interface setting is selected. The second alternate setting (alternate setting 1) is used for normal operation, and shall include one bulk IN endpoint and one bulk OUT endpoint.

The interface descriptors for alternate settings 0 and 1 shall have *bInterfaceSubClass* set to 0, and *bInterfaceProtocol* set to 01h (see section 5.3).

6.4 Extended NCM Feature Descriptors

If Bit D7 of the *bmNetworkCapabilities* field in the NCM Functional Descriptor is set (see Table 6-2), the NCM function is conforming to version 1.1 of this specification. Such a function may support any subset of extended capabilities. The capabilities are grouped according to functional relationships, and a specific type of extended descriptor is used to advertise the supported capabilities in each group. The fields of each descriptor are detailed in section 8. The host may query the set of supported capabilities by issuing a *GetExtendedCapabilities* command, as described in section 8.2.3. In response to this command, the NCM function shall return the general extended capability descriptor described in Table 8-3, followed by the feature descriptors it supports (a subset of Table 6-3).

Table 6-3: List of NCM Extended Descriptors

Descriptor Type	Reference
NCM_MEDIUM_HANDLING	Table 8-6
NCM_WAKE	Table 8-15
NCM_PRESENCE_OFFLOAD	Table 8-25
NCM_TRANSMIT_OFFLOAD	Table 8-33
NCM_RECEIVE_OFFLOAD	Table 8-37

7   NCM 1.0 Communications Class Specific Messages

7.1   Overview

A Communications Interface shall support the standard requests defined in [USBCORE].   In addition, an NCM Communications Interface shall support class- and subclass-specific requests and notifications. These are used to manage the function.

It is an error to send requests to an NCM Data Interface. Device behavior in this case is not specified.

7.2   Network Control Model Requests

Table 7-1 lists the class-specific requests that are valid for an NCM Communications Interface. This table includes those defined in [USBECM12]. Commands marked as "required" must be implemented by any conforming NCM function. Commands marked as "optional" must be implemented in some circumstances; see the Reference section for each command for details. For a description of NCM 1.1 specific requests see Section 8.2.

Table 7-1: Networking Control Model Requests

Request	Description	Req'd/Opt	reference
SendEncapsulatedCommand	Issues a command in the format of the supported control protocol. The intent of this mechanism is to support networking functions (e.g. host-based cable modems) that require an additional vendor-defined interface for media specific hardware configuration and management.	Optional	[USBCDC12]
GetEncapsulatedResponse	Requests a response in the format of the supported control protocol.	Optional	[USBCDC12]
SetEthernetMulticastFilters	Controls the receipt of Ethernet frames that are received with "multicast" destination addresses.	Optional	[USBECM12]
SetEthernetPowerManagement-PatternFilter	Some hosts are able to conserve energy and stay quiet in a "sleeping" state while not being used. NCM functions may provide special pattern filtering hardware that enables the function to wake up the attached host on demand when something is attempting to contact the host (e.g. an incoming web browser connection). This command allows the host to specify the filter values that detect these special frames.	Optional	[USBECM12]
GetEthernetPowerManagement-PatternFilter	Retrieves the status of the above power management pattern filter setting	Optional	[USBECM12]
SetEthernetPacketFilter	Controls the types of Ethernet frames that are to be received via the function.	Optional	[USBECM12]
GetEthernetStatistic	Retrieves Ethernet statistics such as frames transmitted, frames received, and bad frames received.	Optional	[USBECM12]
GetNtbParameters	Requests the function to report parameters that characterize the Network Control Block	Required	7.2.1
GetNetAddress	Requests the current EUI-48 network address	Optional	7.2.2
SetNetAddress	Changes the current EUI-48 network address	Optional	7.2.3
GetNtbFormat	Get current NTB Format	Optional	7.2.4
SetNtbFormat	Select 16 or 32 bit Network Transfer Blocks	Optional	7.2.5
GetNtbInputSize	Get the current value of maximum NTB input size	Required	7.2.6

Request	Description	Req'd/Opt	reference
SetNtbInputSize	Selects the maximum size of NTBs to be transmitted by the function over the bulk IN pipe.	Required	7.2.7
GetMaxDatagramSize	Requests the current maximum datagram size (see 3.6 and 4.7)	Optional	7.2.8
SetMaxDatagramSize	Sets the maximum datagram size to a value other than the default (see 3.6 and 4.7)	Optional	7.2.9
GetCrcMode	Requests the current CRC mode	Optional	7.2.10
SetCrcMode	Sets the current CRC mode	Optional	7.2.11

Table 7-2 describes the request codes that are used in the Networking Control Model Subclass, including those defined in [USBECM12]. See Table 8-2 for description of NCM 1.1 specific codes.

Table 7-2: Class-Specific Request Codes for Network Control Model subclass

Request	Value
SET_ETHERNET_MULTICAST_FILTERS	40h
SET_ETHERNET_POWER_MANAGEMENT_PATTERN FILTER	41h
GET_ETHERNET_POWER_MANAGEMENT_PATTERN FILTER	42h
SET_ETHERNET_PACKET_FILTER	43h
GET_ETHERNET_STATISTIC	44h
GET_NTB_PARAMETERS	80h
GET_NET_ADDRESS	81h
SET_NET_ADDRESS	82h
GET_NTB_FORMAT	83h
SET_NTB_FORMAT	84h
GET_NTB_INPUT_SIZE	85h
SET_ NTB_ INPUT_SIZE	86h
GET_MAX_DATAGRAM_SIZE	87h
SET_MAX_DATAGRAM_SIZE	88h
GET_CRC_MODE	89h
SET_CRC_MODE	8Ah
RESERVED for NCM 1.1	8Bh-8Fh
RESERVED (future use)	90h-9Fh

7.2.1 GetNtbParameters

This request retrieves the parameters that describe NTBs for each direction. In response to this request, the function shall return these elements as listed in Table 7-3.

bmRequestType	bRequestCode	wValue	wIndex	wLength	Data
10100001B	GET_NTB_PARAM-ETERS	zero	NCM Communications Interface	Number of bytes to read	NTB Parameter Structure (Table 7-3)

The returned NTB Parameter Structure is defined in Table 7-3.

Table 7-3: NTB Parameter Structure

Offset	Field	Size	Value	Description
0	wLength	2	Number	Size of this structure, in bytes = 1Ch.
2	bmNtbFormatsSupported	2	Bitmap	Bit 0: 16-bit NTB supported (set to 1) Bit 1: 32-bit NTB supported Bits 2 – 15: reserved (reset to zero; must be ignored by host)
4	dwNtbInMaxSize	4	Number	IN NTB Maximum Size in bytes
8	wNdpInDivisor	2	Number	Divisor used for IN NTB Datagram payload alignment
10	wNdpInPayloadRemainder	2	Number	Remainder used to align input datagram payload within the NTB: (Payload Offset) mod ($wNdpInDivisor$) = $wNdpInPayloadRemainder$
12	wNdpInAlignment	2	Number	NDP alignment modulus for NTBs on the IN pipe. Shall be a power of 2, and shall be at least 4.
14	-reserved	2	Zero	Padding, shall be transmitted as zero by function, and ignored by host.
16	dwNtbOutMaxSize	4	Number	OUT NTB Maximum Size
20	wNdpOutDivisor	2	Number	OUT NTB Datagram alignment modulus
22	wNdpOutPayloadRemainder	2	Number	Remainder used to align output datagram payload offsets within the NTB: (Payload Offset) mod ($wNdpOutDivisor$) = $wNdpOutPayloadRemainder$
24	wNdpOutAlignment	2	Number	NDP alignment modulus for use in NTBs on the OUT pipe. Shall be a power of 2, and shall be at least 4.
26	wNtbOutMaxDatagrams	2	Number	Maximum number of datagrams that the host may pack into a single OUT NTB. Zero means that the device imposes no limit.

To get the full response, the host should set *wLength* to at least 1Ch. The function shall never return more than 1Ch bytes in response to this command.

All NCM functions shall support 16-bit NTBs. Therefore, bit 0 of *bmNtbFormatsSupported* shall always be set to 1. NCM functions may support 32-bit NTBs. If 32-bit NTBs are supported, then GetNtbFormat and SetNtbFormat must be supported.

7.2.2 GetNetAddress

This request returns the function's current EUI-48 station address.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_NET_ADDRESS	Zero	NCM Communications Interface	Number of bytes to read	The EUI-48 current address, in network byte order

To get the entire network address, the host should set *wLength* to at least 6. The function shall never return more than 6 bytes in response to this command.

7.2.3 SetNetAddress

This request sets the function's current EUI-48 station address. It does not change the function's permanent EUI-48 station address, which is given by field *iMACAddress* in the Ethernet Functional Descriptor (see [USBECM12], section 5.4).

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_NET_ADDRESS	Zero	NCM Communications Interface	6	The EUI-48 address, in network byte order

The host shall set *wLength* to 6. The function shall return an error response (a STALL PID) if wLength is set to any other value.

The function resets its EUI-48 station address to the permanent address, as governed by events outside the scope of this command. See section 7.1 for details.

The host shall only send this command while the NCM Data Interface is in alternate setting 0.

7.2.4 GetNtbFormat

This request returns the NTB data format currently being used by the function.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_NTB_FORMAT	Zero	NCM Communications Interface	Number of bytes to read	The NTB format code (2 bytes, little-endian), as defined under *wValue* in *SetNtbFormat* (7.2.5).

To get the full response, the host should set *wLength* to at least 2. The function shall never return more than 2 bytes in response to this command.

This command must be supported by the function if it declares support for an NTB size other than 16bit in *bmNtbFormatsSupported*.

If the function does not support NTB sizes other than 16bit, or if the function is operating in NCM 1.1 mode, then the host shall not issue this command to the function. The function shall return an error response (a STALL PID) if it receives this command while operating in NCM 1.1 mode.

7.2.5 SetNtbFormat

This request selects the format of NTB to be used for NTBs transmitted from the function to the host. The host must choose one of the available choices from the *bmNtbFormatsSupported* bitmap element from the *GetNtbParameters* command response (Table 7-3).

The command format uses the same format, with a single choice selected.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_NTB_FORMAT	NTB Format Selection: 0000h: NTB-16 0001h: NTB-32 All other values are reserved.	NCM Communications Interface	0	None

The host shall only send this command while the NCM Data Interface is in alternate setting 0, and the function is operating in NCM 1.0 mode. Refer to section 2.4 for additional details.

The function's NTB format setting may be changed by events beyond the scope of this command; see section 7.1 for details.

If the value passed in *wValue* is not supported, the function shall return an error response (a STALL PID) and shall not change the format it is using to send and receive NTBs.

This command must be supported by the function if it declares support for an NTB size other than 16bit in *bmNtbFormatsSupported*.

If the function does not support NTB sizes other than 16bit, or if the function is operating in NCM 1.1 mode, then the host shall not issue this command to the function. The function shall return an error response (a STALL PID) if it receives this command while operating in NCM 1.1 mode.

7.2.6 GetNtbInputSize

This request returns NTB input size currently being used by the function.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_NTB_INPUT_SIZE	Zero	NCM Communications Interface	Number of bytes to read	The NTB input size structure, as defined in *SetNtbInputSize* (6.2.7).

To get the NTB input size, the host should set *wLength* to at least 4. To get the full NTB input size structure, the host should set *wLength* to at least 8. If bit D5 is set in field *bmNetworkCapabilities* of the function's NCM Functional Descriptor, the function shall never return more than 8 bytes in response to this command. If bit D5 is reset, the function shall never return more than 4 bytes in response to this command. The fields in the input size structure are returned in little-endian order

7.2.7 SetNtbInputSize

This request selects the maximum size of NTB that the device is permitted to send to the host, and optionally the maximum number of datagrams that the device is permitted to encode into a single NTB.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_NTB_INPUT_SIZE	Zero	NCM Communications Interface	4 or 8	If *wLength* is 8, then this is the NTB Input Size Structure. If *wLength* is 4, then this is the *dwNtbInMaxSize* field of the NTB Input Size Structure.

Table 7-4. NTB Input Size Structure

Offset	Field	Size	Value	Description
0	dwNtbInMaxSize	4	Number	IN NTB Maximum size in bytes. The host shall select a size that is at least 2048, and no larger than the maximum size permitted by the function, according to the value given in the NTB Parameter Structure
4	wNtbInMaxDatagrams	2	Number	Maximum number of datagrams within the IN NTB. Zero means no limit.
6	reserved	2	Number	Shall be transmitted as zero and ignored upon receipt.

If bit D5 is set in the *bmNetworkCapabilities* field of function's NCM Functional Descriptor, the host may set *wLength* either to 4 or to 8. If *wLength* is 4, the function shall assume that *wNtbInMaxDatagrams* is to be set to zero. If *wLength* is 8, then the function shall use the provided value as the limit. The function shall return an error response (a STALL PID) if *wLength* is set to any other value.

If bit D5 is reset in the *bmNetworkCapabilities* field of the function's NCM Functional Descriptor, the host shall set *wLength* to 4. The function shall return an error response (a STALL PID) if *wLength* is set to any other value.

If the value passed in the data phase is not valid, or if *wLength* is not valid, the function shall return an error response (a STALL PID) and shall not change the value it uses for preparing NTBs.

7.2.8 GetMaxDatagramSize

In response to this request, the function returns the current maximum datagram size on the external medium, which may be different from the maximum datagram size in an NTB. Refer to sections 3.6 and 4.7 for details.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_MAX_DATAGRAM_-SIZE	Zero	NCM Communications Interface	Number of bytes to read	The current maximum datagram size, in little endian order (2 bytes).

To get the full response, the host should set *wLength* to at least 2. The function shall never return more than 2 bytes in response to this command.

7.2.9 SetMaxDatagramSize

This request selects the maximum datagram size on the external medium, which may be different from the maximum datagram size in an NTB. Refer to sections 3.6 and 4.7 for details.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_MAX_DATAGRAM_-SIZE	Zero	NCM Communications Interface	2	Maximum datagram size, in bytes, in little-endian order

The host shall select a size that is at least 1514, and no larger than the maximum size permitted by the function, according to the value given by *wMaxSegmentSize* in the Ethernet Networking Functional Descriptor ([USBECM12], 5.4).

The host shall set *wLength* to 2. The function shall return an error response (a STALL PID) if *wLength* is set to any other value. If the value passed in the data phase is not valid, the function shall return an error response (a STALL PID) and shall not change the value it uses as current maximum datagram size.

The function's maximum datagram size is set to a default value by events outside the scope of this command; see section 7.2 for details.

7.2.10 GetCrcMode

This request returns the currently selected CRC mode for NTBs formatted by the function.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_CRC_MODE	Zero	NCM Communications Interface	Number of bytes to read	The current CRC mode, in little endian order (2 bytes).

To get the full response, the host should set *wLength* to at least 2. The function shall never return more than 2 bytes in response to this command.

Two values are possible. The function shall return 0000h if CRCs are not being appended to datagrams. The function shall return 0001h if CRCs are being appended to datagrams.

7.2.11 SetCrcMode

This request controls whether the function will append CRCs to datagrams when formatting NTBs to be sent to the host.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_CRC_MODE	CRC mode: 0000h: CRCs shall not be appended 0001h: CRCs shall be appended All other values are reserved.	NCM Communications Interface	0	None

If the value passed in *wValue* is not valid, the function shall return an error response (a STALL PID) and shall not change the CRC mode.

The function's CRC mode is set to a default value by events outside the scope of this command; see section 7.1 for details.

7.3   Network Control Model Extended Features

Extended features are supported as of Revision 1.1 of this specification. Refer to section 8.2 for the description of currently supported features.

7.4   Network Control Model Notifications

[USBCDC12] defines the common Communication Class notifications that the function uses to notify the host of events related to that function. These notifications are sent over the interrupt IN pipe that is in the Communications Interface.

The class-specific notifications valid for an NCM Communication Interface are listed in Table 7-5.

Table 7-6 lists the corresponding notification codes. For convenience, we repeat the codes defined in [USBCDC12].

Certain operational and sequencing requirements, which are more specific than the requirements in [USBCDC12], are imposed for notifications (refer to section 9.1).

Table 7-5: Networking Control Model Notifications

Notification	Description	Req'd/Opt	reference
NetworkConnection	Reports whether or not the physical layer (modem, Ethernet PHY, etc.) link is up.	Required	[USBCDC12]
ResponseAvailable	Notification to host to issue a *GetEncapsulatedResponse* request.	Optional (Mandatory if *SetEncapsulatedCommand* and *GetEncapsulatedResponse* are supported)	[USBCDC12]
ConnectionSpeedChange	Reports a change in upstream or downstream speed of the networking connection.	Required	[USBCDC12]

Table 7-6: Class-Specific Notification Codes for Networking Control Model subclass

Request	Value
NETWORK_CONNECTION	00h
RESPONSE_AVAILABLE	01h
CONNECTION_SPEED_CHANGE	2Ah

8   NCM 1.1 Communications Class Specific Messages

8.1   Overview

NCM 1.1 functions shall support all the existing NCM 1.0 requests, as detailed in section 7.2, and additionally the NCM 1.1-specific requests required to inquire about extended capabilities, enable the extended features and query their status, where applicable.

8.2   Network Control Model Requests

Table 8-1 lists the new requests that are valid for an NCM 1.1 Communications Interface. The NCM 1.0 requests (Table 7-1) are implicitly supported by any NCM 1.1 function. The conditions under which implementation of each request is mandatory are listed in the Req'd/Opt column.

The host cannot predict the effect of any of the requests in Table 8-1 if the function does not support NCM 1.1 extended requests (see section 6.4). The rest of this section assumes that NCM 1.1 extended requests are supported.

Table 8-1: Networking Control Model Requests (NCM 1.1)

Request	Description	Req'd/Opt	reference
GetExtendedCapabilityMode	Queries the function current mode (NCM 1.0 or NCM 1.1)	Mandatory for NCM 1.1 functions	8.2.1
SetExtendedCapabilityMode	Sets the function to NCM 1.1 mode	Mandatory for NCM 1.1 functions	8.2.2
GetExtendedCapabilities	Returns a description of the extended capabilities supported by this device	Mandatory for NCM 1.1 functions	8.2.3
GetExtendedFeature	Request status information about a specific extended feature	Mandatory if NCM Extended Feature Functional Descriptor is present	8.2.4
SetExtendedFeature	Send control information related to a specific extended feature	Mandatory if NCM Extended Feature Functional Descriptor is present	8.2.5

Table 8-2 describes the requests new to Networking Control Model Subclass, revision 1.1.

Table 8-2: New Class-Specific Request Codes for NCM 1.1

Request	Value
GET_EXTENDED_CAPABILITY_MODE	8Bh
SET_EXTENDED_CAPABILITY_MODE	8Ch
GET_EXTENDED_CAPABILTIES	8Dh
GET_EXTENDED_FEATURE	8Eh
SET_EXTENDED_FEATURE	8Fh
RESERVED (future use)	90h-9Fh

8.2.1 GetExtendedCapabilityMode

This request queries the current status of the extended capability mode of an NCM 1.1 function. If bit D7 is set in field *bmNetworkCapabilities* of the function's NCM Functional Descriptor (section 6.2.1), the host may issue this request to retrieve the current operating mode of the function (see section 2.4).

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_CAPABILITY_MODE	Zero	NCM Communications Interface	Number of bytes to read	The current capability mode, in little endian order (2 bytes). 0000h: NCM 1.0 mode. NCM 1.0 data path (section 3) and NCM 1.0 notifications (section 7.4) are used. 0001h: NCM 1.1 mode. NCM 1.1 data path (section 4) and NCM 1.1 notifications (section 8.4) are used. NCM 1.1 requests (section 8.2) are available. Other values are reserved.

8.2.2 SetExtendedCapabilityMode

This request enables the NCM 1.1 extended capability mode. If bit D7 is set in field *bmNetworkCapabilities* of the function's NCM Functional Descriptor (section 6.2.1), the host may issue this request to switch the function to NCM 1.1 mode (see section 2.4).

Enabling NCM 1.1 extended capability mode also switches the data transport to NCM 1.1 (section 4) and enables NCM 1.1 notifications (section 8.4). Therefore, in order to avoid ambiguity in the data transport, the host shall only send this command while the NCM Data Interface is in alternate setting 0.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_CAPABILITY_MODE	Capability mode, in little endian order (2 bytes), as described in 8.2.1.	NCM Communications Interface	0	None

8.2.3 GetExtendedCapabilities

This request fetches a description of the extended capabilities implemented by the function. If bit D7 is set in field *bmNetworkCapabilities* of the function's NCM Functional Descriptor (section 6.2.1), the host may issue this request to query the supported capabilities.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-CAPABILTIES	Zero	NCM Communications Interface	Number of bytes to read	Capability data, see Table 8-3

The capability data is a collection of NCM extended capability descriptors that is similar to the USB configuration descriptor. The collection always starts with a header descriptor that includes the overall length. The host can read the header and get the overall length, and it can use that overall length to read the entire collection of descriptors.

Table 8-3. NCM Extended Capability Descriptor

Offset	Field	Size	Value	Description
0	bLength	1	5	Size of descriptor in bytes
1	bDescriptorType	1	Constant	CS_INTERFACE (0x24)
2	bDescriptorSubtype	1	Constant (1Dh)	NCM Extended Capability Descriptor subtype, as defined in Table 5-4.
3	wTotalLength	2	Number	Length of this descriptor in bytes, including all of the feature descriptors.

Table 8-4. NCM Extended Feature Descriptor

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of descriptor in bytes
1	bDescriptorType	1	Constant	CS_INTERFACE (0x24)
2	bDescriptorSubtype	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	bNcmFeatureSelector	1	Number	Indicates the feature described (Table 8-5).
4..bLength-1				Optional feature-specific data.

Table 8-5: *bNcmFeatureSelector* value for supported extended features

Feature	bNcmFeatureSelector
NCM_MEDIUM_HANDLING	01h
NCM_WAKE	02h
NCM_PRESENCE_OFFLOAD	03h
NCM_TRANSMIT_OFFLOAD	04h
NCM_RECEIVE_OFFLOAD	05h

8.2.4 GetExtendedFeature

This request fetches feature-specific status information from the function.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-FEATURE	Depends on extended feature selector	Bits 0..7: NCM Communications Interface Bits 8..15: extended feature selector	Number of bytes to read	Feature-specific data

See section 8.3 for discussion of specific extended features.

The host shall only send this request if it has previously sent *SetExtendedCapabilityMode* (section 8.2.1) to enable NCM 1.1 mode. If the feature selected by bits 8..15 of *wIndex* is not supported in the function (as reported by *GetExtendedCapabilities*, section 8.2.3), the function shall return an error response (a STALL PID) and shall otherwise not change state.

8.2.5 *SetExtendedFeature*

This request sets feature-specific information to the function.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_-FEATURE	Depends on extended feature selector	Bits 0..7: NCM Communications Interface Bits 8..15: extended feature selector	Number of bytes to write	Feature-specific data

See section 8.3 for discussion of specific extended features.

The host shall only send this request if it has previously sent *SetExtendedCapabilityMode* (section 8.2.1) to enable NCM 1.1 mode. If the feature selected by bits 8..15 of *wIndex* is not supported by the function (as reported by *GetExtendedCapabilities*, section 8.2.3), the function shall return an error response (a STALL PID) and shall otherwise not change state.

8.3   NCM 1.1 Extended Features

8.3.1   Medium Handling

The NCM function shall specify the medium it supports. Each function shall support a single medium. Medium-specific parameters may be presented by each medium type. A Medium is identified by:

- Type (see Table 8-8)
- Speed
- Type-specific parameters

The device can send a *UnifiedMediumNotification* to the Host (see section 8.4.1). The unified medium notification would take the place of existing *ConnectionSpeedChange* and *NetworkConnection* (section 7.4).

If the host sends the GET_EXTENDED_CAPABILITIES request, and the NCM_MEDIUM_HANDLING feature is supported by the NCM function, the extended feature descriptor in Table 8-6 shall be included in the returned descriptor collection.

Table 8-6. NCM Medium Handling Feature Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	14	Size of descriptor in bytes
1	*bDescriptorType*	1	Constant	CS_INTERFACE (0x24)
2	*bDescriptorSubtype*	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	*bNcmFeatureSelector*	1	NCM_MEDIUM_HANDLING	Medium Handling feature
4..13	*stSupportedMedium*	10	Medium structure	See Table 8-7

The Medium structure consists of the following fields:

Table 8-7. Medium structure for Medium handling

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	Medium Type (see Table 8-8)
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Medium Notification Supported[2]
2	*dwSpeed*	4	Number (little endian)	A value of all zeros – indicates no medium. A value of all ones – indicates auto-select / auto-negotiation. Otherwise - Medium speed in Kbps (kilobits per second).
6	*bmMediumParameters*	4	Bit field	D31:D0 - Medium Type specific data

---

[2]*UnifiedMediumNotification* is mandatory for NCM 1.1 functions, therefore Bit D0 shall be set to 1 in the Medium structure. See 0 and 9.2 for additional information.

Table 8-8: NCM Medium Type Codes

Value	Name	Description
00h	NCM_MEDIUM_VIRTUAL_WIRE	Virtual Wire
01h	NCM_MEDIUM_ETHERNET	[IEEE802.3] Ethernet
02h-FFh	Reserved	Reserved for future use

Table 8-9. Medium structure for Virtual Wire (Medium Type 00h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_VIRTUAL_WIRE**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Medium Notification Supported
2	*dwSpeed*	4	Number (little endian)	A value of all ones – indicates that the function supports auto-negotiation. Otherwise - Medium speed in Kbps (kilobits per second).
6	*bmMediumParameters*	4	Bit field	D31:D2 - RESERVED D1 - **Can provide WAN to Host**. An NCM function with multiple physical interfaces (e.g., a cellular modem) may use this bit to indicate its ability to provide Internet connectivity to the host. D0 – **Expects WAN from Host**. An NCM function (e.g., an embedded device) may use this bit to indicate to the Host that it expects the Host to bridge it to the Internet via another interface. Note: It is expected that at most one of the bits D0, D1 will be set for a given NCM function.

Table 8-10. Medium structure for Ethernet (Medium Type 01h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_ETHERNET**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Medium Notification Supported
2	*dwSpeed*	4	Number (little endian)	A value of all ones – indicates auto-select / auto-negotiation. Otherwise - Highest supported Medium speed in Kbps (kilobits per second).
6	*bmMediumParameters*	4	Bit field	Supported capabilities. A value of '1' in the corresponding bit indicates that a capability is supported. A value of '0' indicates that it is not supported. D31 – [IEEE802.3AZ] Energy Efficient Ethernet (EEE) D30 – 802.3x flow control Tx Pause frames D29 – 802.3x flow control Rx Pause frames D28:D14 - RESERVED D13 – 400 Gbps link speed D12 – 200 Gbps link speed D11 – 100 Gbps link speed D10 – 50 Gbps link speed D9 – 40 Gbps link speed D8 – 25 Gbps link speed D7 – 10 Gbps link speed D6 – 5 Gbps link speed D5 – 2.5 Gbps link speed D4 – 1 Gbps link speed D3 – 100 Mbps Full Duplex link speed D2 – 100 Mbps Half Duplex link speed D1 – 10 Mbps Full Duplex link speed D0 – 10 Mbps Half Duplex link speed

If *dwSpeed* is set to all ones (auto-negotiation is supported), then bits D13:D0 indicate the subset of speeds that the device can advertise via selective speed advertisement during the [IEEE802.3] auto-negotiation process. Otherwise, they indicate the subset of speeds that the function can be forced to.

The host may send a SET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_MEDIUM_HANDLING to configure a specific medium.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_-FEATURE	0	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_MEDIUM_HANDLING	Number of bytes to write	Medium structure

If the medium specified in the *Data* field is not the supported medium, the function shall return an error response (a STALL PID) and shall otherwise not change state.

Table 8-11. SET_EXTENDED_FEATURE Medium Structure for Virtual Wire (Medium Type 00h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_VIRTUAL_WIRE**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Enable Medium Notification
2	*dwSpeed*	4	Number (little endian)	A value of all zeros – Disable medium A value of all ones – Enable medium All other values are reserved and shall be interpreted by the Device function as the Host's intent to Enable medium.
6	*bmMediumParameters*	4	Bit field	D31:D0 - RESERVED

Table 8-12. SET_EXTENDED_FEATURE Medium structure for Ethernet (Medium Type 01h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_ETHERNET**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Enable Medium Notification
2	*dwSpeed*	4	Number (little endian)	A value of all zeros – Disable medium A value of all ones – Enable medium with auto-negotiation according to the enabled speed bits in bmMediumParameters. Otherwise - Medium speed in Kbps (kilobits per second).
6	*bmMediumParameters*	4	Bit field	Specifies the capabilities to enable/advertise. A value of '1' in the corresponding bit indicates that a capability is enabled. A value of '0' indicates that it is disabled. The interpretation of the bits is according to the value of bmMediumParameters in Table 8-10.

The host may send a GET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_MEDIUM_HANDLING to get the current medium status.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-FEATURE	0	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_MEDIUM_HANDLING	Number of bytes to read	Medium structure

If the medium specified in the *Data* field is not the supported medium, the function shall return an error response (a STALL PID) and shall otherwise not change state.

Table 8-13. GET_EXTENDED_FEATURE Medium Structure for Virtual Wire (Medium Type 00h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_VIRTUAL_WIRE**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Medium Notification Enabled
2	*dwSpeed*	4	Number (little endian)	A value of all zeros – Medium is disabled A value of all ones – Medium is enabled All other values are reserved and shall be interpreted by the Host as the medium being enabled.
6	*bmMediumParameters*	4	Bit field	D31:D2 - RESERVED D1 - **Can provide WAN to Host**. An NCM function with multiple physical interfaces (e.g., a cellular modem) may use this bit to indicate its ability to provide Internet connectivity to the host. D0 – **Expects WAN from Host**. An NCM function (e.g., an embedded device) may use this bit to indicate to the Host that it expects the Host to bridge it to the Internet via another interface. Note: It is expected that at most one of the bits D0, D1 will be set for a given NCM function.

Table 8-14. GET_EXTENDED_FEATURE Medium structure for Ethernet (Medium Type 01h)

Offset	Field	Size	Value	Description
0	*bMediumType*	1	Type code	**NCM_MEDIUM_ETHERNET**
1	*bmFeatureFlags*	1	Bit field	D7:D1 – Reserved D0 – Medium Notification Enabled
2	*dwSpeed*	4	Number (little endian)	Zero – no link Non-zero – Link speed in Kbps (kilobits per second)
6	*bmMediumParameters*	4	Bit field	Specifies the active/advertised capabilities. A value of '1' in the corresponding bit indicates that a capability is active. A value of '0' indicates that it is inactive. The interpretation of the bits is according to the value of bmMediumParameters in Table 8-10. The link speed advertising bits are valid only if the function supports [IEEE802.3] auto-negotiation, as indicated by the dwSpeed field in the NCM_MEDIUM_HANDLING feature descriptor of Table 8-10. Otherwise, these bits are meaningless and shall be set to 0.

8.3.2 Function Wake

The NCM function may support a number of wake capabilities typical for network devices.

If the host sends the GET_EXTENDED_CAPABILITIES request, and the NCM_WAKE feature is supported by the NCM function, one or more extended feature descriptors in Table 8-15 shall be included in the returned descriptor collection.

Table 8-15. NCM Function Wake Feature Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Varies	Size of descriptor in bytes
1	*bDescriptorType*	1	Constant	CS_INTERFACE (0x24)
2	*bDescriptorSubtype*	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	*bNcmFeatureSelector*	1	NCM_WAKE	Function Wake feature
4	*bWakeType*	1	NCM_WAKE_TYPE	See Table 8-16
5..	*stWakeTypeData*	0..	Capability-specific structure	Included only if required for the specific value of *bWakeType*. See Table 8-17, Table 8-18, Table 8-19.

Table 8-16: NCM_WAKE_TYPE Codes

Value	Name	Description
00h	NCM_WAKE_MEDIA_CONNECT	Wake on media connect
01h	NCM_WAKE_MEDIA_DISCONNECT	Wake on media disconnect
02h	NCM_WAKE_MAGIC_PACKET	Wake on magic packet
03h	NCM_WAKE_PATTERN_FILTER	Wake on pattern filter (Table 8-17)
04h	NCM_WAKE_ANY_PACKET	Wake on any packet (Table 8-18)
05h	NCM_WAKE_TCP_UDP_PORT	Wake on TCP/UDP destination port (Table 8-19)
06h	NCM_WAKE_MDNS_RESPONDER	Wake on mDNS responder
07h-FEh	Reserved	Reserved for future use
FFh	NCM_WAKE_GET_REASON	Not a wake type. Indicates a query to get the reason for the most recently occurred wake.

Some NCM_WAKE_TYPE codes define an additional type-specific data structure (*stWakeTypeData*), which shall be appended to the NCM Function Wake Feature Descriptor using the associated NCM_WAKE_TYPE.

Table 8-17. Capability-specific data structure for Pattern filter wake

Offset	Field	Size	Value	Description
0	*wNumPatterns*	2	Number	Maximum number of pattern filters supported for wake up
2	*wMaxPatternLength*	2	Number	Maximum pattern length for pattern filters (little-endian)

Table 8-18. Capability-specific data structure for Any packet wake

Offset	Field	Size	Value	Description
0	*bmSupportedPacketTypes*	2	Bit field	D15:D5 – Reserved D4 – Function supports wake on broadcast MAC address[3] D3 – Function supports wake on any multicast MAC address[3] D2 – Function supports wake on multicast packets matching the configured multicast filters (see *SetEthernetMulticastFilters* in Table 7-1) D1 – Function supports wake on any unicast MAC address[3] D0 – Function supports wake on directed unicast packets (destination address matching the station address of this device)

Table 8-19. Capability-specific data structure for destination TCP/UDP Port wake

Offset	Field	Size	Value	Description
0	*bNumTcpPorts*	1	Number	Maximum number of TCP destination ports supported for wake up
1	*bNumUdpPorts*	1	Number	Maximum number of UDP destination ports supported for wake up

The host may send a SET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_WAKE to configure a specific wake capability.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_-FEATURE	WakeMode Bits 0..7: Reserved Bits 8..15: NCM_WAKE_TYPE (except NCM_WAKE_GET_REASON)	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_WAKE	Number of bytes to write	As described in Table 8-20

If the *wValue* is not one of the defined values, or if the capability represented by *wValue* is not one of the supported capabilities, according to Table 8-15, or if the wake-specific data structure requests unsupported sub-capabilities, the function shall return an error response (a STALL PID) and shall otherwise not change state.

---

[3] In all cases, the destination MAC address of the incoming packet (as defined in [IEEE802.3]) is compared to the relevant packet type / filter. A MAC address is unicast if the least-significant bit of the first byte is '0', otherwise it is considered multicast. The broadcast MAC address is FF:FF:FF:FF:FF:FF.

Table 8-20. Extended Feature set/get data field for NCM_WAKE

NCM_WAKE_TYPE value	Description
NCM_WAKE_MEDIA_CONNECT, NCM_WAKE_MEDIA_DISCONNECT, NCM_WAKE_MAGIC_PACKET, or NCM_WAKE_MDNS_RESPONDER	If disabled: a single WORD (little endian) with value 0x00 If enabled: a single WORD (little endian) with value 0x01
NCM_WAKE_PATTERN_FILTER	Configuration structure for the filter number in WakeType (Table 8-21)
NCM_WAKE_ANY_PACKET	Configuration structure (Table 8-22)
NCM_WAKE_TCP_UDP_PORT	Configuration structure (Table 8-23)
NCM_GET_WAKE_REASON	Wake reason and wake packet structure (Table 8-24)

Table 8-21. Extended Feature set/get structure for Wake Pattern Filter

Offset	Field	Size	Value	Description
0	*bmFeatureFlags*	2	Bit field	D15:D1 – Reserved D0 – Enable Wake on pattern filter
2	*wFilterNumber*	2	Number	Number of pattern filter being configured. Must be between 0 and *wNumPatterns-1* (see Table 8-17).
4	*patternLength*	2	Number	Length (bytes) of the pattern. The total length of the structure is derived from this field as (patternLength + ⌈ patternLength / 8 ⌉), since the byte pattern is followed by a bit mask.
6.. patternLength+5	*patternBytes*	*patternLength*	Array	Array of byte values to perform pattern matching on
patternLength + 6	*patternMask*	⌈ patternLength / 8 ⌉ (*patternLength* / 8, rounded up)	Bit field	Each byte of *patternMask* contains 8 masking bits, where each one of them represents whether or not the associated byte of *patternBytes* should be compared with what is seen on the media by the networking device. The least significant bit (D0) of the first Mask byte is associated with the first byte of the pattern, which starts at the Destination Address (DA) of the Ethernet frame. If *wPatternLength* is not a multiple of 8, the corresponding highest order bits in the last byte of *patternMask* should be set to 0.

Table 8-22 Extended Feature set/get structure for Any Packet Wake

Offset	Field	Size	Value	Description
0	*bmFeatureFlags*	2	Bit field	D15:D1 – Reserved D0 – Enable Wake on any packet
2	*bmEnabledWakes*	2	Bit field	D15:D5 – Reserved D4 – Wake on broadcast D3 – Wake on any multicast D2 – Wake on matched multicast D1 – Wake on any unicast D0 – Wake on directed unicast (See Table 8-18 for additional definitions)

Table 8-23. Extended Feature set/get structure for TCP/UDP port wake

Offset	Field	Size	Value	Description
0	*bmFeatureFlags*	2	Bit field	D15:D1 – Reserved D0 – Enable Wake on TCP/UDP port
2	*bNumTcpPorts*	1	Number	Number **T** of supported TCP destination ports for wake up
3	*bNumUdpPorts*	1	Number	Number **U** of supported UDP destination ports for wake up
4 .. 2*T+3	*arrTcpPorts*	2 * *bNumTcpPorts*	Array of numbers	Array of TCP destination ports to enable wake on, in network order
2*T+4 .. 2 * (T+U)+3	*arrUdpPorts*	2 * *bNumUdpPorts*	Array of numbers	Array of UDP destination ports to enable wake on, in network order

The host may send a GET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_WAKE either to query the current configuration for a given wake type, or to query the wake reason and associated packet (if applicable), for the most recent wake. The host shall use the value of NCM_WAKE_TYPE to specify what it is querying for.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-FEATURE	WakeType Bits 0..7: *bFilterNumber* if wake type is NCM_WAKE_ PATTERN_FILTER. Otherwise: reserved Bits 8..15: NCM_WAKE_TYPE to retrieve wake configuration or reason.	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_WAKE	Number of bytes to read	As described in Table 8-20

Table 8-24. Wake Reason and Wake Packet Structure

Offset	Field	Size	Value	Description
0	*wWakeReason*	2	Bit field	Bits 0..7: *bFilterNumber* if NCM_WAKE_TYPE is equal to NCM_WAKE_ PATTERN_FILTER. Otherwise: reserved Bits 8..15: NCM_WAKE_TYPE (Table 8-16)
2	*wakePacketLength*	2	Number	Length of wake packet. If *bmWakeReason* is 0, or if no packet is associated with the wake (for example – wake on link status change), this field shall be 0 as well.
4..	*wakePacketData*	*wakePacketLength*	Array	Array of packet data bytes, with the least significant byte corresponding to the first byte of the Destination Address (DA) of the Ethernet frame.

8.3.2.1 Magic Packet definition

A magic packet is defined as any valid [IEEE802.3] Ethernet frame, where the [IEEE802.3] 6-byte MAC address of the station is repeated 16 (sixteen) times, consecutively, in the payload. In particular, the length of the magic packet is always at least 114 bytes (14 bytes Ethernet header + 96 bytes payload + 4 bytes CRC). The destination address of the frame might be the MAC address of the station or the broadcast address (FF:FF:FF:FF:FF:FF).

8.3.2.2 Additional considerations for device wake

When wake on specific types of packets is enabled, the NCM function is still expected to filter packets not explicitly addressed to the host. Packets whose destination MAC address does not match that of the host shall not wake the system, even if they would otherwise qualify for one of the armed wake filters, with the following exceptions:

- Wake on any packet (bit D4 in *bmSupportedWakeModes*) with one of the promiscuous modes enabled (Table 8-22)
- Wake on pattern filter (bit D3 in *bmSupportedWakeModes*) where the pattern for the destination MAC address (first 6 bytes) is explicitly set to a value different from the station address.
- Wake on magic packet (bit D2 in *bmSupportedWakeModes*), which may be sent as a broadcast packet, as described in 8.3.2.1.

8.3.3 Presence Offloads

The NCM function may support a number of presence offload capabilities (also referred to as "protocol offload" or "proxy offload"). Presence offload allows the NIC hardware / firmware to automatically respond to certain types of packets, while the host system remains in a state of low power. Presence offload capabilities are typically used in conjunction with wakeup capabilities, in order to maintain visible presence of the "sleeping" host in the network, but only waking it upon reception of packets targeting the host directly.

If the host sends the GET_EXTENDED_CAPABILITIES request, and the NCM_PRESENCE_OFFLOAD feature is supported by the NCM function, one or more extended feature descriptors in Table 8-25 shall be included in the returned descriptor collection.

Table 8-25. NCM Presence Offload Feature Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	Varies	Size of descriptor in bytes
1	*bDescriptorType*	1	Constant	CS_INTERFACE (0x24)
2	*bDescriptorSubtype*	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	*bNcmFeatureSelector*	1	NCM_PRESENCE_OFFLOAD	Presence Offload feature
4	*bOffloadType*	1	NCM_PRESENCE_OFFLOAD_TYPE	See Table 8-26
5..	*stOffloadTypeData*	0..	Capability-specific structure	See Table 8-27, Table 8-28, Table 8-29

Table 8-26: NCM_PRESENCE_OFFLOAD_TYPE Codes

Value	Name	Description
00h	NCM_PRESENCE_OFFLOAD_ARP	ARP offload (IPv4 Address Resolution Protocol, RFC-826)
01h	NCM_PRESENCE_OFFLOAD_NS	NS offload (IPv6 Neighbor Solicitation, RFC-4861)
02h	NCM_PRESENCE_OFFLOAD_MDNS	mDNS responder offload (Multicast DNS, RFC-6762)
03h-FFh	Reserved	Reserved for future use

Some NCM_PRESENCE_OFFLOAD_TYPE codes define an additional type-specific data structure which shall be appended to the NCM Presence Offload Feature Descriptor using the associated NCM_PRESENCE_OFFLOAD_TYPE.

Table 8-27. Capability-specific data structure for ARP offload

Offset	Field	Size	Value	Description
0	*bMaxIpv4Addresses*	1	Number	Maximum number of IPv4 addresses supported for the offload.
1	*bMaxLocalIpv4Addresses*	1	Number	Maximum number of local IPv4 addresses. Must be smaller than or equal to *bMaxIpv4Addresses.* An address is considered local if it belongs to the device managed by the NCM function. The NCM function shall only use 'Local' addresses for ARP offload. 'Other' addresses (those which are not 'Local'), shall be used by the NCM function for mDNSResponder name conflict resolution.

Table 8-28. Capability-specific data structure for NS offload

Offset	Field	Size	Value	Description
0	*bMaxIpv6Addresses*	1	Number	Maximum number of IPv6 addresses supported for the offload.
1	*bMaxLocalIpv6Addresses*	1	Number	Maximum number of local IPv6 addresses. Must be smaller than or equal to *bMaxIpv6Addresses.* An address is considered local if it belongs to the device managed by the NCM function. The NCM function shall only use 'Local' addresses for NS offload. 'Other' addresses (those which are not 'Local'), shall be used by the NCM function for mDNSResponder name conflict resolution.

Table 8-29. Capability-specific data structure for mDNS offload

Offset	Field	Size	Value	Description
0	*wMaxNumResourceRecords*	2	Number	Maximum number of RFC-1035 Domain Name Resource Records (RRs) that the NCM function can support.
2	*wMaxResourceRecordSize*	2	Number	Maximum total buffer size for all Resource Records (RRs).

The host may send a SET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_PRESENCE_OFFLOAD to configure one of the supported presence offloads.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_-FEATURE	OffloadType Bits 0..7: Reserved Bits 8..15: NCM_PRESENCE_ OFFLOAD_TYPE	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_PRESENCE_ OFFLOAD	Number of bytes to write	Configuration data for the specified offload (Table 8-30, Table 8-31, Table 8-32)

If the *wValue* is not one of the defined values, or if the capability represented by *wValue* is not one of the supported capabilities, according to Table 8-25, the function shall return an error response (a STALL PID) and shall otherwise not change state.

Table 8-30. Extended Feature set/get data structure for ARP Offload

Offset	Field	Size	Value	Description
0	bmFeatureFlags	2	Bit field	D15:D1 – Reserved D0 – Enable ARP offload
2	bNumIpv4	1	Number	Total number of IPv4 addresses in this offload
3	bNumLocalIpv4	1	Number	Number of local IPv4 addresses in this offload. *bNumLocalIpv4 <= bNumIpv4.* The first *bNumLocalIpv4* addresses shall be treated as local.
4	stMacAddress	6	[IEEE802.3] MAC address	The link layer address of the network device. Must be set to a valid [IEEE802.3] unicast MAC address, or to zero. If set to zero, the function shall use its current MAC address (see 7.2.2).

*10 .. bNumIpv4 * 8 + 9*	arrIpv4Addresses	*bNumIpv4 * 8*	Array of IPv4 [RFC791] (address,mask) pairs	Array of pairs:
				  32-bit IPv4 address, in network order
				  32-bit IPv4 mask to apply to the address, in network order. An unused mask may be set to all ones, or all zeros.

Table 8-31. Extended Feature set/get data structure for NS Offload

Offset	Field	Size	Value	Description
0	bmFeatureFlags	2	Bit field	D15:D1 – Reserved D0 – Enable NS offload
2	bNumIpv6	1	Number	Total number of IPv6 addresses in this offload
3	bNumLocalIpv6	1	Number	Number of local IPv6 addresses in this offload. b*NumLocalIpv6 <= bNumIpv6.* The first *bNumLocalIpv6* addresses shall be treated as local.
4	stMacAddress	6	[IEEE802.3] MAC address	The link layer address of the network device. This field is optional. If set to zero, the function shall use its current MAC address (see 7.2.2).

*10 .. bNumIpv6 * 32 +9*	arrIpv6Addresses	*bNumIpv6 * 32*	Array of IPv6 [RFC8200] (address, mask) pairs	Array of pairs:
				  128-bit IPv6 address, in network order
				  128-bit IPv6 mask to apply to the address, in network order. An unused mask may be set to all ones, or all zeros.

**Table 8-32. Extended Feature set/get data structure for mDNS Offload**

Offset	Field	Size	Value	Description
0	*bmFeatureFlags*	2	Bit field	D15:D1 – Reserved D0 – Enable mDNS offload
2	*wResourceRecordLength*	2	Number	Total number of bytes present in the *arrResourceRecords* field
4	*wResourceRecordOffset*	2	Number	Number of prepended bytes in the *arrResourceRecords* field[4]
6	*wNumResourceRecords*	2	Number	Number of individual Resource Records present in *arrResourceRecords*
8…	*arrResourceRecords*	Varies	Array of RR structs	Array of Resource Record (RR) structures, as described in RFC-1035, with *wResourceRecordOffset* bytes prepended (see example in Figure 8-1).

The host may send a GET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_PRESENCE_OFFLOAD to retrieve the previously configured settings for one of the supported presence offloads.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-FEATURE	OffloadType Bits 0..7: Reserved Bits 8..15: NCM_PRESENCE_ OFFLOAD_TYPE	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_PRESENCE_ OFFLOAD	Number of bytes to read	Configuration data for the specified offload (Table 8-30, Table 8-31, Table 8-32)

If the *wValue* is not one of the defined values, or if the capability represented by *wValue* is not one of the supported capabilities, according to Table 8-25, the function shall return an error response (a STALL PID) and shall otherwise not change state.

The configuration data returned by this request shall be equivalent to the configuration data that was previously sent via the SET_EXTENDED_FEATURE request for the respective offload.
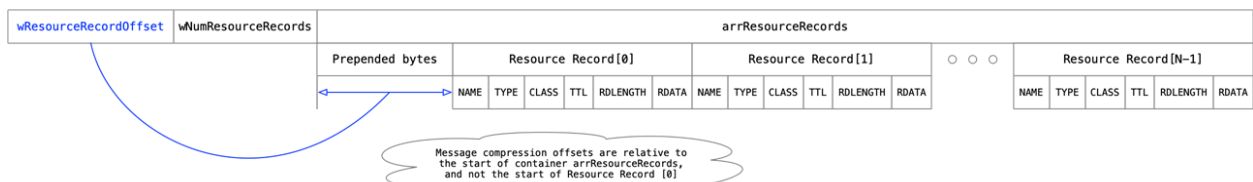
Figure 8-1 – Example of SET_EXTENDED_FEATURE data structure for mDNS Offload

---

[4] In some mDNS responder applications, resource records are not self-contained but are generated in the context of a containing message (such as a DNS message). According to RFC-1035, the message may be compressed, and compression offsets are relative to those of the containing message, and not the start of the first resource record. Because the practical length of the prepended bytes is small, it is convenient for the Host to send these extra bytes in the USB transfer. The total size of *arrResourceRecords* includes those bytes and must not exceed the capabilities which are advertised by the function in *wMaxResourceRecordSize* (Table 8-29). See Figure 8-1 for an example, where *wNumResourceRecords* = N, and *wResourceRecordOffset* is the number of prepended bytes in *arrResourceRecords*.

8.3.4 Transmit Offloads

The NCM function may support transmit offload capabilities, where the hardware or firmware can update existing fields in the datagram, insert new fields, and perform automatic segmentation of large datagrams to MTU-sized chunks.

If the host sends the GET_EXTENDED_CAPABILITIES request, and the NCM_TRANSMIT_OFFLOAD feature is supported by the NCM function, then one or more transmit offload feature descriptors (Table 8-33) shall be included in the returned descriptor collection.

Table 8-33. NCM Transmit Offload Feature Descriptor

Offset	Field	Size	Value	Description
0	bLength	1	Varies	Size of descriptor in bytes
1	bDescriptorType	1	Constant	CS_INTERFACE (0x24)
2	bDescriptorSubtype	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	bNcmFeatureSelector	1	NCM_TRANSMIT_OFFLOAD	Transmit Offload feature
4	bTransmitOffloadType	1	NCM_TRANSMIT_OFFLOAD_TYPE	See Table 8-34
5..	stTransmitOffloadData	0..	Capability-specific structure	See Table 8-35 and Table 8-36

Table 8-34: NCM_TRANSMIT_OFFLOAD_TYPE Codes

Value	Name	Description
00h	NCM_TRANSMIT_OFFLOAD_CHECKSUM	Checksum offload
01h	NCM_TRANSMIT_OFFLOAD_SEGMENTATION	Segmentation offload (large send)
02h	NCM_TRANSMIT_OFFLOAD_VLAN	[IEEE802.1Q] VLAN tag insertion
03h-FFh	Reserved	Reserved for future use

Some NCM_TRANSMIT_OFFLOAD_TYPE codes define an additional type-specific data structure which shall be appended to the NCM Transmit Offload Feature Descriptor using the associated NCM_TRANSMIT_OFFLOAD_TYPE.

Table 8-35. Capability-specific data structure for transmit checksum offload

Offset	Field	Size	Value	Description
0	*bmSupportedOffloads*	2	Bit field	Checksum offloads supported by the function: D15:D4 – Reserved D3 – UDP checksum offload D2 – TCP checksum offload D1 – IPv4 with options checksum offload D0 – IPv4 checksum offload

Table 8-36. Capability-specific data structure for transmit segmentation offload

Offset	Field	Size	Value	Description
0	*bmSupportedOffloads*	2	Bit field	Supported segmentation offloads: D15:D4 – Reserved D3 – UDP/IPv6 segmentation offload D2 – UDP/IPv4 segmentation offload D1 – TCP/IPv6 segmentation offload D0 – TCP/IPv4 segmentation offload
2	*wMaximumOffloadSize*	2	Number	Maximum supported frame size for Large Send Offload, in KiB. An NCM function may support a maximum size lower than the theoretical maximum of the underlying network protocols. The Host shall not send datagrams with the Length field in the Transmit meta-info structure (4.4.1) set higher than *wMaximumOffloadSize.*

Specific transmit offloads may be enabled or disabled on an individual datagram (NDPX) level, as described in section 4.4.1. Therefore, GET_EXTENDED_FEATURE (section 8.2.4) and SET_EXTENDED_FEATURE (section 8.2.5) commands are not applicable to transmit offloads. If the host sends either of these, with *bNcmFeatureSelector* set to NCM_TRANSMIT_OFFLOAD, the function shall return an error response (a STALL PID) and shall otherwise not change state.

8.3.4.1 Checksum offloads and pseudo-header checksums

Layer 4 (TCP and UDP) checksums are computed over the entire header, packet payload, and a selected subset of fields from the Layer 3 (IPv4 or IPv6) header, as described in [RFC768], [RFC793], [RFC2460]. When Layer 4 checksum offload is enabled, software shall populate the checksum field with the partial checksum calculation over the Layer 3 pseudo-header fields. The NCM function shall offload checksum calculation over the rest of the data (Layer 4 header, and packet payload).

8.3.5 Receive Offloads

The NCM function may support common receive offload capabilities, such as automatic verification of checksums, [IEE802.1Q] VLAN tag stripping, RSS hash computation (section 10) and Large Receive Offload (coalescing multiple Ethernet frames into a single large datagram).

If the host sends the GET_EXTENDED_CAPABILITIES request, and the NCM_RECEIVE_OFFLOAD feature is supported by the NCM function, then one or more receive offload feature descriptors (Table 8-37) shall be included in the returned descriptor collection.

Table 8-37. NCM Receive Offload Feature Descriptor

Offset	Field	Size	Value	Description
0	bLength	1	Varies	Size of descriptor in bytes
1	bDescriptorType	1	Constant	CS_INTERFACE (0x24)
2	bDescriptorSubtype	1	Constant (1Eh)	NCM Extended Feature Descriptor subtype, as defined in Table 5-4.
3	bNcmFeatureSelector	1	NCM_RECEIVE_OFFLOAD	Receive Offload feature
4	bReceiveOffloadType	1	NCM_RECEIVE_OFFLOAD_TYPE	See Table 8-38
5..	stReceiveOffloadData	0..	Capability-specific structure	See Table 8-39, Table 8-40, Table 8-41

Table 8-38: NCM_RECEIVE_OFFLOAD_TYPE Codes

Value	Name	Description
00h	NCM_RECEIVE_OFFLOAD_CHECKSUM	Checksum verification offload
01h	NCM_RECEIVE_OFFLOAD_COALESCING	Large receive (receive coalescing) offload
02h	NCM_RECEIVE_OFFLOAD_VLAN	[IEEE802.1Q] VLAN tag stripping
03h	NCM_RECEIVE_OFFLOAD_RSS	RSS hash computation
04h	NCM_RECEIVE_OFFLOAD_STORE_BAD_PACKETS	Store bad packets (failed Ethernet CRC check)
05h-FFh	Reserved	Reserved for future use

Some NCM_RECEIVE_OFFLOAD_TYPE codes define an additional type-specific data structure which shall be appended to the NCM Receive Offload Feature Descriptor using the associated NCM_RECEIVE_OFFLOAD_TYPE.

Table 8-39. Capability-specific data structure for receive checksum offload

Offset	Field	Size	Value	Description
0	*bmSupportedOffloads*	2	Bit field	Checksum offloads supported by the function: D15:D4 – Reserved D3 – UDP checksum verification offload D2 – TCP checksum verification offload D1 – IPv4 with options checksum verification offload D0 – IPv4 checksum verification offload

Table 8-40. Capability-specific data structure for receive coalescing offload

Offset	Field	Size	Value	Description
0	*bmSupportedOffloads*	2	Bit field	Supported receive coalescing offloads: D15:D4 – Reserved D3 – UDP/IPv6 receive coalescing D2 – UDP/IPv4 receive coalescing D1 – TCP/IPv6 receive coalescing D0 – TCP/IPv4 receive coalescing
2	*wMaximumOffloadSize*	2	Number	Maximum supported frame size for Large Receive Coalescing, in KiB.

Table 8-41. Capability-specific data structure for RSS hash offload

Offset	Field	Size	Value	Description
0	*bmSupportedRssHashes*	2	Bit field	Supported RSS hashes. A bit of '1' indicates that the specific hash is supported. If all bits are '0', the NCM function does not support RSS. See section 10 for additional details. D15:D9 - Reserved D8 - UDP/IPv6 with Extensions D7 – UDP/IPv6 D6 – UDP/IPv4 D5 – TCP/IPv6 with Extensions D4 – TCP/IPv6 D3 – IPv6 with Extensions D2 – IPv6 D1 – TCP/IPv4 D0 – IPv4

The host may send a SET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_RECEIVE_OFFLOAD to configure one of the supported presence offloads.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001B	SET_EXTENDED_-FEATURE	OffloadType Bits 0..7: Reserved Bits 8..15: NCM_RECEIVE_ OFFLOAD_TYPE	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_RECEIVE _OFFLOAD	Number of bytes to write	Configuration data for the specified offload (Table 8-42, Table 8-43, Table 8-44, Table 8-45)

Table 8-42. Data structure for checksum verification offload configuration

Offset	Field	Size	Value	Description
0	*bmChecksumOffloads*	2	Bit field	Checksum verification offloads to enable. The bits are interpreted according to the description of *bmSupportedOffloads* in Table 8-39.

Table 8-43. Data structure for receive coalescing offload configuration

Offset	Field	Size	Value	Description
0	*bmChecksumOffloads*	2	Bit field	Receive coalescing offloads to enable. The bits are interpreted according to the description of *bmSupportedOffloads* in Table 8-40.

Table 8-44. Data structure for RSS offload configuration

Offset	Field	Size	Value	Description
0	*bmRssHashes*	2	Bit field	RSS hashes to enable. The bits are interpreted according to the description of *bmSupportedRssHashes* in Table 8-41.
2	*arrRssSecretKey*	40	Array	RSS secret key, as described in section 10.1

Table 8-45. Data structure for other receive offload configuration

Offset	Field	Size	Value	Description
0	*bmEnabledOffload*	2	Bit field	D15:D1 – Reserved D0 – '1': Enable offload, '0': Disable offload

If the any feature that is unsupported according the advertised Receive Offload capabilities is enabled, the function shall return an error response (a STALL PID) and shall otherwise not change state.

The host may send a GET_EXTENDED_FEATURE request with *bNcmFeatureSelector* set to NCM_RECEIVE_OFFLOAD to retrieve the previously configured settings for one of the supported presence offloads.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001B	GET_EXTENDED_-FEATURE	OffloadType Bits 0..7: Reserved Bits 8..15: NCM_RECEIVE_OFFLOAD_TYPE	Bits 0..7: NCM Communications Interface Bits 8..15: NCM_RECEIVE_OFFLOAD	Number of bytes to read	Configuration data for the specified offload (Table 8-42, Table 8-43, Table 8-44, Table 8-45)

8.4   NCM 1.1 Notifications

When running in NCM 1.0 mode, an NCM 1.1 function shall support all standard NCM notifications as described in section 7.4. When running in NCM 1.1 mode, an NCM function shall support the extended notifications below:

Table 8-46: NCM 1.1 Extended Notifications

Notification	Description	Req'd/Opt	reference
UnifiedMediumNotification	Reports link status and speed	Required	8.4.1

Table 8-47: Class-Specific Notification Codes for NCM 1.1

Request	Value
UNIFIED_MEDIUM_STATE	2Bh

8.4.1   UnifiedMediumNotification

This notification allows the device to inform the host-networking driver that the link state or link speed have changed. It may be used as a substitute for the separate *NetworkConnection* and *ConnectionSpeedChange* notifications.

bmRequestType	bNotificationCode	wValue	wIndex	wLength	Data
10100001B	UNIFIED_MEDIUM_STATE	Zero	Interface	10	Medium structure

The medium structure included in the data of this notification command is the same structure that would be returned in a response to a GET_EXTENDED_FEATURE request for NCM_MEDIUM_HANDLING. Refer to section 8.3.1 for details.

9   Operational Constraints

9.1   Notification Sequencing

NCM functions are required to send *ConnectionSpeedChange* and *NetworkConnection* notifications in a specific order. To simplify the coding of host device drivers, functions that are going to send a *NetworkConnection* notification with wValue == 0001h must first send a *ConnectionSpeedChange* notification that indicates the connection speed that will be in effect when the new connection takes effect.

This sequencing is justified as follows. If the *ConnectionSpeedChange* follows a *NetworkConnection* notification, then the host driver cannot signal network connection with the correct speed until the *ConnectionSpeedChange* is received. This delay may introduce latency between bus events and system events, or may cause host system overhead due to a spurious change in speed. If the function signals the connection speed first, then the host driver will know the signaling speed at the time the network connection becomes valid.

Developers should be aware that this sequence of notifications (speed before connection) is different than the sequence specified in [USBECM12].

9.2   Using Alternate Settings to Reset an NCM Function

To place the network aspects of a function in a known state, the host shall:

- select alternate setting 0 of the NCM Data Interface (this is the setting with no endpoints). This can be done explicitly using *SetInterface*, or implicitly using *SetConfiguration*. See [USBCORE] for details.

- select the NCM operational parameters by sending commands to the NCM Communication Interface, then

- select the second alternate interface setting of the NCM Data Interface (this is the setting with a bulk IN endpoint and a bulk OUT endpoint).

Whenever alternate setting 0 is selected by the host, the function shall:

- flush function buffers

- reset the packet filter to its default state

- clear all multicast address filters

- clear all power filters set using *SetEthernetPowerManagementPatternFilter*

- reset statistics counters to zero

- restore its Ethernet address to its default state

- reset its IN NTB size to the value given by field *dwNtbInMaxSize* from the NTB Parameter Structure

- reset the NTB format to NTB-16

- reset the current Maximum Datagram Size to a function-specific default. If *SetMaxDatagramSize* is not supported, then the maximum datagram size shall be the same as the value in *wMaxSegmentSize* of the Ethernet Networking Functional Descriptor (see Table 6-1 and [USBECM12], 5.4). If *SetMaxDatagramSize* is supported by the function, then the host must either query the function to determine the current effective maximum datagram size, or must explicitly set the maximum datagram size. If the host wishes to set the Maximum Datagram Size, it may do so prior to selecting the second alternate interface setting of the data interface. Doing so will ensure that the change takes effect prior to send or receiving data.

- reset CRC mode so that the function will not append CRCs to datagrams sent on the IN pipe

- reset NTB sequence numbers to zero

- If the function is NCM 1.1-capable and was operating in NCM 1.1 mode, it shall additionally:

  - disable and reset any NCM 1.1 extended features
  - return to NCM 1.0 operation mode (see 2.4)

When the function is in NCM 1.0 mode, and the host selects the second alternate interface setting of the NCM Data Interface, the function shall perform the following actions in the following order.

- If connected to the network, the function shall send a *ConnectionSpeedChange* notification to the host indicating the current connection speed.

- Whether connected or not, the function shall then send a *NetworkConnection* notification to the host, with *wValue* indicating the current state of network connectivity

When the function is in NCM 1.1 mode (set previously by the *SetExtendedCapabilityMode* request), and the host selects the second alternate interface setting of the NCM Data Interface, the function shall perform the following actions in the following order.

- Send a *UnifiedMediumNotification* indicating the current state of network connectivity, and the connection speed, if applicable.

9.3   Remote Wakeup and Network Traffic

USB Devices containing NCM functions may support remote wakeup. There are two general situations in which remote wakeup may be used:

1. to awaken a link that has been selectively suspended ("link suspend");
2. to awaken the USB host from a system suspend state ("system suspend").

A function cannot distinguish between cases 1 and 2 at the bus level. In case 1, the function should signal remote wakeup whenever traffic is received from the network or when an indication should be passed to the host. In case 2, the link should be awakened only when it's time to wake up the host system. NCM functions distinguish between these two cases based on whether the host has used *SetEthernetPowerManagementPatternFilter* to set up an active power management filter. If *GetEthernetPowerManagementPatternFilter* would return 0001h at the time that the device or function enters suspend, then the function shall follow the rules for system suspend given in 9.3.2. Otherwise, the function shall follow the rules for link suspend given in 9.3.1.

NCM 1.1 functions shall rely on the wake modes configured via the Function Wake extended feature (section 8.3.2) instead of the *SetEthernetPowerManagementPatternFilter* command.

9.3.1 Remote Wakeup Rules for Link Suspend

In this case, if remote wakeup is enabled and if the device is suspended, the NCM function shall request a remote wakeup whenever:

1. Enough time has elapsed since suspend to allow remote wakeup to be signaled according to [USBCORE], AND

2. The function has not yet signaled remote wakeup or received remote wakeup from the upstream hub, AND

3. A non-zero alternate setting was selected for the Data Interface prior to suspend, AND

4. EITHER network traffic is available for the host over the bulk IN pipe of the Data Interface, OR notifications are available for the host over the interrupt IN pipe of the Communications Interface, OR the physical medium state has changed and wake on medium state change is enabled (see 8.3.2).

The USB device may define additional remote wakeup conditions. However, these conditions are sufficient to allow the host driver to suspend an NCM function transparently to save system and device power when performance is not critical.

While the link is suspended, functions shall make best efforts to retain network traffic and notifications that cause the wakeup condition, and any traffic or notifications received between the time that remote wakeup is signaled and when the link wakes up.

9.3.2 Remote Wakeup Rules for System Suspend

In this case, if remote wakeup is enabled and the device is suspended, the NCM function shall request a remote wakeup whenever:

1. Enough time has elapsed since suspend to allow remote wakeup to be signaled according to [USBCORE], AND

2. The function has not yet signaled remote wakeup or received remote wakeup from the upstream hub, AND

3. A non-zero alternate setting was selected for the Data Interface prior to suspend, AND

4. A packet matching one of the configured wakeup filters is received from the network, OR the physical medium state has changed and wake on medium state change is enabled (see 8.3.2)

While the link is suspended, functions shall observe changes in network connectivity and connection speed, but these changes shall not cause a remote wakeup to be signaled. Upon resume, if the network connectivity or connection speeds have changed compared to the state when the link was suspended, the function shall send the appropriate notification(s) to inform the host of the new network connection state (see 7.4 and 8.4).

Network traffic received while the function is suspended, other than packets matching the pre-configured wakeup filters, shall be discarded by the function.

9.4   Using the Interface Association Descriptor

USB Device containing NCM functions may include an Interface Association Descriptor (IAD) to indicate that the Communication Class interface and the Data Class Interface are to be treated as a single function. (This is an alternative to using the WMC WHCM Union descriptor.)

If an IAD is used, the IAD should appear before the Communication Class Interface. The IAD's *bFunctionClass* shall be set to 02h (Communication Class), which is the same value used in *bInterfaceClass* of the Communication Class interface; *bFunctionSubClass* shall be set to 0Dh (Network Control Model), which is the same value used in *bInterfaceSubClass* of the Communication Class interface; and *bFunctionProtocol* shall be set to the same value as is used in *bInterfaceProtocol* of the Communication Class interface for the function. In accordance with the IAD ECN, *bInterfaceCount* shall be set to 2, and *bFirstInterface* shall be set to the interface number of the Communication Class Interface.

When using an IAD, the general order of descriptors for an NCM function shall be:

- Interface Association Descriptor (IAD)

- Communications Class Interface Descriptor (interface n)

- Functional descriptors for the Communication Class Interface (refer to Section 6.2)

- Endpoint descriptors for the Communication Class Interface

- Data Class Interface (interface n+1, alternate setting 0)

- Functional descriptors for Data Class Interface (interface n+1, alternate setting 0)

- Data Class Interface (interface n+1, alternate setting 1)

- Functional descriptors for Data Class Interface (interface n+1, alternate setting 1)

- Endpoint descriptors for Data Class Interface (interface n+1, alternate setting 1)

10 Appendix - Receive Side Scaling (RSS) Parameters

Based on Microsoft Learn documentation for Windows Drivers. Used with permission.

10.1 Definitions

Receive side scaling (RSS) is a network driver technology that enables the efficient distribution of network receive processing across multiple CPUs in multiprocessor systems. At the core of RSS is a hash function, which is computed by the network device over predefined sections of the received network datagram. The resulting hash value helps the network driver stack select a CPU for processing the packet. The exact mechanism of CPU selection is outside the scope of this document.

The hashing is performed by the Toeplitz hash function, as follows:

1. The overlying driver provides a secret key (K) to the miniport driver for use in the hash calculation. The key is 40 bytes (320 bits) long.

2. Given an input array that contains n bytes, the byte stream is defined as:

```
input[0] input[1] input[2] ... input[n-1]
```

3. The left-most byte is input[0], and the most-significant bit of input[0] is the left-most bit. The right-most byte is input[n-1], and the least-significant bit of input[n-1] is the right-most bit.

4. Given the preceding definitions, the pseudocode for processing a general input byte stream is defined as follows:

```
ComputeHash(input[], n)

    result = 0

    For each bit b in input[] from left to right

    {

        if (b == 1)

            result ^= (left-most 32 bits of K)

        shift K left 1 bit position

    }

    return result
```

10.2 Fields used for hashing

The datagram header fields over wish the hash is computed depend on the datagram type and on the hash types supported by the network device. The supported hash types are specified in the *RSSType* field of the Receive meta-info structure (section 4.4.3, Table 4-6).

Table 10-1: RSS Hash Types

RSSType	Hash Type	Fields
1	IPv4	Source IP address, Destination IP address
2	IPv6	Source IP address, Destination IP address
3	IPv6 with Extensions	Extension IP Source Address (*), Extension IP Destination Address (**)
4	TCP/IPv4	Source IP address, Destination IP address, Source TCP Port, Destination TCP Port
5	TCP/IPv6	Source IP address, Destination IP address, Source TCP Port, Destination TCP Port
6	TCP/IPv6 with Extensions	Extension IP Source Address (*), Extension IP Destination Address (**), Source TCP Port, Source TCP Destination Port
7	UDP/IPv4	Source IP address, Destination IP address, Source UDP Port, Destination UDP Port
8	UDP/IPv6	Source IP address, Destination IP address, Source UDP Port, Destination UDP Port
9	UDP/IPv6 with Extensions	Extension IP Source Address (*), Extension IP Destination Address (**), Source UDP Port, Source UDP Destination Port

(*) Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 Address.

(**) IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 Address.

10.3 Examples

In the examples below, the notation `@m-n` refers to the range of bytes `[m,m+1,…,n]` in the packet.

10.3.1 TCP/IPv4 packets

Concatenate the *SourceAddress*, *DestinationAddress*, *SourcePort*, and *DestinationPort* fields of the packet into a byte array, preserving the order in which they occurred in the packet:

```
Input[12] = @12-15, @16-19, @20-21, @22-23

Result = ComputeHash(Input, 12)
```

10.3.2 IPv6 packets

Concatenate the *SourceAddress* and *DestinationAddress* fields of the packet into a byte array, preserving the order in which they occurred in the packet.

```
Input[36] = @8-23, @24-39

Result = ComputeHash(Input, 32)
```