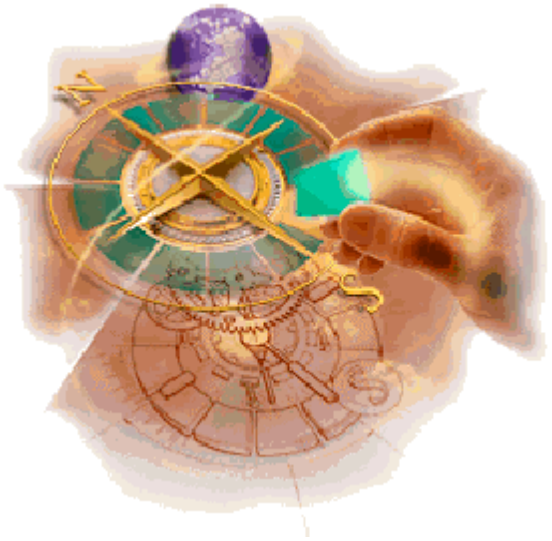
Novell®

ODI SPECIFICATION SUPPLEMENT:

ECB Extensions

Documentation Version 1.10
July 31, 2000



Disclaimer

Novell, Inc. makes no representations or warranties with respect to the content or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of Novell software, at any time, without obligation to notify any person or entity of such changes.

This product may require export authorization from the U.S. Department of Commerce prior to exporting from the U.S. or Canada.

U.S. Patent Nos 5,553,139; 5,553,143, 5,677,851; 5,758,069; 5,784,560; 5,818,936; 5,864,865; 5,903,650; 5,905,860; 5,910,803 and other Patents Pending.

trademarks

Novell and NetWare are registered trademarks of Novell, Inc. in the United States and other countries.

All other company and product names are trademarks or registered trademarks of their respective owners.

Novell, Inc.
1800 South Novell Place
Provo, UT 84606
U.S.A.

www.novell.com

Contents

1 Introduction

1.1 Problem

The basic structure of the current ECB used by ODI in NetWare today has been untouched for nearly ten years since it's introduction with NetWare 3.0. Early on it was realized that the ECB as defined had limited room to expand. In 1993 there was a proposal to modify the ECB to allow for future expandability. That proposal was tabled for a variety of reasons, one of which was that it broke the existing software of the day. Over the years the reserved fields and bits have slowly been defined until there are only a couple of bits left. Today, we are faced with the challenge of moving forward while preserving the past. We are constantly being asked by our partners in the industry to support new technologies in the ODI model, such as VLAN and hardware offloading such as IPSec. Many of these technologies require that additional information must accompany the incoming and outgoing data as it moves up and down the data path.

Unfortunately, as the reserved field and bits in the ECB have been defined, the current ECB does not have the ability to support these technologies.

1.2 Solution

This document proposes extensions to the ECB used by the ODI driver model in order to support these new technologies. This proposal does not change the current ECB structure; instead, it outlines how the current structure can be used to transport the additional information that must accompany the data. If approved, these changes would become part of the next C ODI Specification revision.

1.3 Issues

The following is a list of issues that must be understood when reviewing this proposal.

- All components in the data path must understand the extensions (i.e., bound stacks, prescan stacks, LSL, MSM, TSMs, HSMs, etc.). Constructing a data path that has a mixture of modules that do and do not understand the ECB extensions can be catastrophic.
- NLMs, such as CLIB, that build their own transmit ECBs and NLMs that are ECB-aware will need to be modified to handle the extensions.
- Support for ECB extensions will be provided only in the C ODI specifications. IHVs must have a C HSM in order to support ECB extensions.
- Support for ECB extensions in Protocol stacks will be provided with minimum changes to the Assembly spec, but C Spec support for protocol stacks will be the preferred method of implementation.
- The extensions require an ODI specification revision. This may cause problems with modules that look for a specific version and fail if the version does not match exactly.
- Extensions will be provided only for the NetWare5.x platform.
- Extensions must be extensible to support future technologies.
- IHVs with drivers supporting Hardware Checksum offloading will need to be aware of the changes made to the ECB_StackID and make the appropriate adjustments. (See 2.1 for changes to the Transmit ECB ECB_StackID field.).

- HSM/Hardware that require ECBs to be located in Memory below 16 MB will not be able to support ECB extensions.

2 ECB Auxiliary Data

Many new technologies require the ability to associate additional information with the send and receive data being passed between protocol stacks and drivers. This document will refer to the additional information as auxiliary data.

The ECB header used by ODI in NetWare today has no free space available to place or point to the auxiliary data. This section describes the proposed usage of the Transmit and Receive ECB definitions for including auxiliary data.

2.1 Transmit ECB

A transmit ECB that includes auxiliary data is identified by the TX_AUX_DATA value in the ECB_StackID field of the ECB. When this value is present, the ECB has auxiliary data associated with it.

The last fragment pointed to by the fragment list contains the auxiliary data. The length of the auxiliary data is not reflected in the ECB_DataLength. Also, the ECB_FragmentCount does not include the fragment used for auxiliary data. However, the number of fragments, including the auxiliary data fragment, must not exceed 16. The format of the auxiliary data will be described later in this document.

ECB_StackID

When the ECB is prepared by the Protocol to send a packet, the Protocol typically places its Protocol Stack ID that was assigned by the LSL, when the protocol stack registered, in the ECB_StackID field. The ECB_StackID field of the ECB may hold the Protocol stack ID given when the stack registered with the LSL, or a special set of values to communicate raw sends, priority packet transmission, checksumming generation, or auxiliary data.

The following values are valid for the ECB_StackID field.

0x0000-0x00FF (0-255) Protocol Identification Number assigned by the LSL

0xznFy The ECB_StackID field may contain information concerning the transmit request as defined by 0xznFy.

0xFy Priority Transmission Support. (See Priority Tx Support, page 5-25, ODI Specification: C HSM ODI Spec, and ECB_StackID definition, ODI Specification: Protocol Stacks and MLIDS (C Language) on page A-5) The Fy values are the same as the last byte of the Priority Send values. The values decode as follows:

Raw Sends

The Raw Send designation indicates that the Protocol has generated the full MAC header. The MLID will not change the MAC headers on a Raw Send. Raw Send packets may be checksummed. Fy values for raw sends are defined as follows:

TX_RAW_SEND_PRIORITY_0	0xFF	0 = No Priority
TX_RAW_SEND_PRIORITY_1	0xFE	1 = Lowest Priority
TX_RAW_SEND_PRIORITY_2	0xFD	
TX_RAW_SEND_PRIORITY_3	0xFC	
TX_RAW_SEND_PRIORITY_4	0xFB	
TX_RAW_SEND_PRIORITY_5	0xFA	
TX_RAW_SEND_PRIORITY_6	0xF9	
TX_RAW_SEND_PRIORITY_7	0xF8	7 = Highest Priority

Normal Sends

TX_SEND_PRIORITY_0	0xF7	0 = No Priority
TX_SEND_PRIORITY_1	0xF6	1 = Lowest Priority
TX_SEND_PRIORITY_2	0xF5	
TX_SEND_PRIORITY_3	0xF4	
TX_SEND_PRIORITY_4	0xF3	
TX_SEND_PRIORITY_5	0xF2	
TX_SEND_PRIORITY_6	0xF1	
TX_SEND_PRIORITY_7	0xF0	7 = Highest Priority

0xFFFy Values from 0xFFF0 – 0xFFFF (zn = 0xFF) indicate legacy raw and normal send definitions for the ECB_StackID.

0xzn The **z** and **n** nibbles of the zn byte provide additional information about the transmit request. The z and n nibbles are defined as follows:

0xz The **z** nibble of the **zn** byte has a pattern of 10xx. The following values are defined for the z:

TX_NO_CHECKSUM	0x8	No Checksum generation on this packet.
TX_TRANSPORT_CHECKSUM	0x9	Generate the Transport Layer Checksum (TCP, UDP, ICMP, RSVP)
TX_NETWORK_CHECKSUM	0xA	Generate the Network Layer Checksum (Ipv4)

0xn The **n** nibble of the zn byte indicates a value of 0x0 – 0xF. The following values are defined for the n nibble of the zn byte:

Reserved	0x0	Reserved
TX_AUX_DATA	0x1	Auxiliary Data is present
Reserved	0x2 – 0xF	Reserved

2.2 Receive ECB

A receive ECB that includes auxiliary data is identified by the PA_RX_AUX_DATA bit in the ECB_PreviousLink field of the ECB. A pointer to the auxiliary data is also provided in the ECB_FragmentCount + 1 fragment pointer of the ECB. As the received packet does not start in the first byte of the receive buffer, the offset to the start must be indicated by setting RCBReserved[28] (for ECB Aware drivers it is the Protocol workspace (PWs_i32val[0])) to the number of bytes the packet must be shifted for alignment issues plus 8 bytes for the Aux Block fragment pointer.

The auxiliary data will always be aligned on the first dword boundary following the valid data in the fragment structure. This means that there may be up to three bytes of NULL padding between the beginning of the auxiliary data and the end of the received packet. The CHSM/MLID must always calculate the alignment of the beginning of the auxiliary data. The length of the auxiliary data is not reflected in the ECB_DataLength field. Neither does the ECB_FragmentCount include the fragment used for auxiliary data. However, the number of fragments, including the auxiliary data fragment, must not exceed 16. Typically, the number of fragments for receives is always one. The format of the auxiliary data is described later in this document.

The Aux block fragment structure is the same as any data fragment block. A pointer and a length to the AUX block must be provided in this field.

ECB_PreviousLink

This field is typically used as a back link to manage a list of ECBs. The current owner of the ECB uses this field. When an ECB is returned from the MLID containing a received packet, this field contains the received packet error status and other information defined as follows:

Bit Value	Description
PAE_CRC_BIT	CRC error (for example, Frame Check Sequence (FCS) error).
PAE_CRC_ALIGN_BIT	CRC / Frame Alignment error.
PAE_RUNT_PACKET_BIT	Runt packet.
PAE_TOO_BIG_BIT	The packet is larger than allowed by media.
PAE_NOT_ENABLED_BIT	Received packet for a frame type not supported. For example, Logical Board not registered for the frame type of the received packet. A board number associated with the physical adapter is placed in the lookahead structure.
PAE_MALFORMED_BIT	The packet is malformed. For example, the size of the packet is smaller than the minimum size for Media Header (e.g., incomplete MAC Header). Another example is if the contents of the length field in an Ethernet 802.3 header are larger than the total packet size.
PA_NO_COMPRES_BIT	Do not decompress the received packet.
PA_NONCAN_ADDR_BIT	The address present in ECB_ImmediateAddress is in non-canonical format.
PAE_TRANS_PROT_CHKSUM_ERR	The transport layer protocol checksum failed validation: TCP, UDP, RSVP, ICMP.
PAE_NET_PROT_CHKSUM_ERR	The network layer protocol checksum failed validation: Ipv4.
PA_TRANS_PROT_CHKSUM	The transport layer protocol checksum validation was performed: TCP, UDP, RSVP, ICMP.
PA_NET_PROT_CHKSUM	The network layer protocol checksum validation was performed: Ipv4.

PA_RX_AUX_DATA	Auxiliary Data is associated with the received packet.

If no error bits are set, the packet was received without error and the data can be used. All undefined bits are cleared.

2.3 Auxiliary Data format

Auxiliary data is a physically contiguous chunk of memory that is organized into one or more auxiliary data blocks. The ODI specification defines the format for the auxiliary data block header. The author of a particular auxiliary data block defines the payload. If a piece of software does not recognize a particular data block when parsing the auxiliary data, then it should skip over it and check the next block.

The total size of all the auxiliary data blocks must not exceed the total size of Auxiliary Data space. For transmits and receives, the total size of the auxiliary data space is stored in the first auxiliary data block. The size of the auxiliary data space on transmits is also reflected in the FragmentLength of the FRAGMENT_STRUCT associated with Auxiliary Data. (See Appendix A, ODI Specification: Protocol Stacks and MLIDs (C Language), Spec v1.11 – Doc v1.22)

Each auxiliary data block consists of an auxiliary data block header (AUX_DATA_BLOCK_HDR) and the payload as defined by the author. The AUX_DATA_BLOCK_HDR must be the first element in the auxiliary data block. The auxiliary data block header is defined as follows:

```
typedef struct _AUX_DATA_BLOCK_HDR
{
        UINT32          auxType;
        UINT32          auxVersion;
        UINT32          auxLength;
        UINT32          auxPayloadVersion;
}AUX_DATA_BLOCK_HDR;
```

auxType	A unique ID that identifies this auxiliary data block. All auxiliary data block auxType values are assigned by Novell. Third parties that author auxiliary data blocks must register them with Novell in order to get a unique type value assigned for their auxiliary data block.
auxVersion	The version of the AUX_DATA_BLOCK definition. Set this field to AUX_DATA_BLOCK_HDR_VERSION.
auxLength	Total length in bytes of the auxiliary data block. This length includes the size of the AUX_DATA_BLOCK_HDR plus the size of the payload.
auxPayloadVersion	The version of the payload. The author of the auxiliary data block determines the version of the payload.

The first auxiliary data block must be of *auxType* AUX_DATA_START. The last auxiliary data block must be *auxType* AUX_DATA_END. Figure 1 shows the format of the Auxiliary Data.

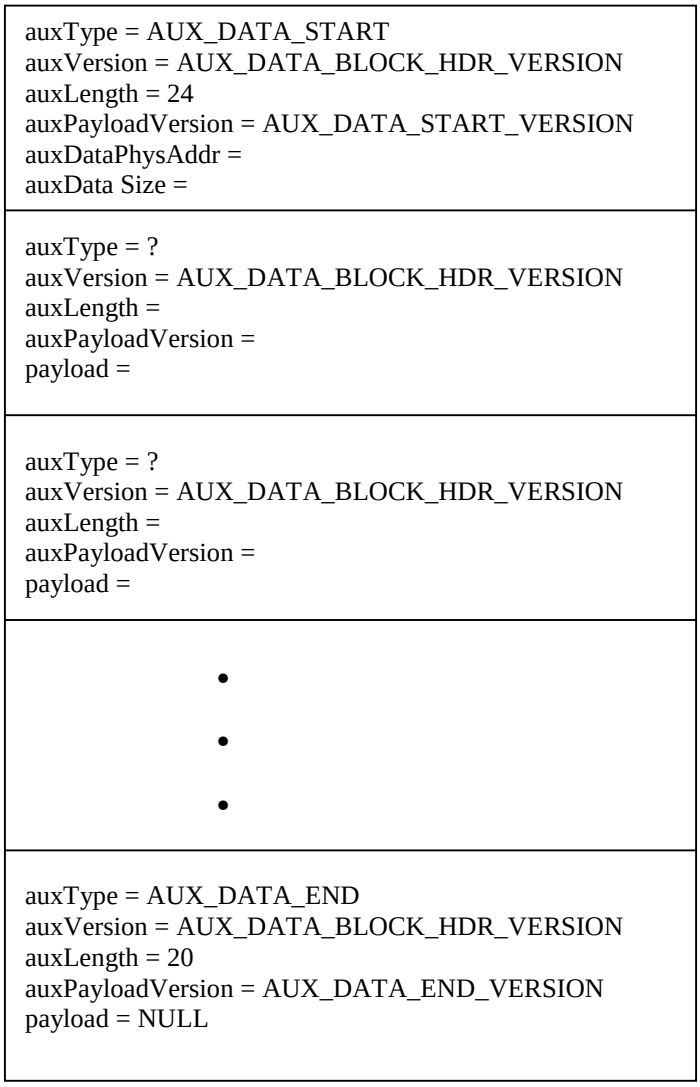


Figure 1 Auxiliary Data Format

2.3.1 Auxiliary Data Start

AUX_DATA_START is the first auxiliary data block in an Auxiliary Data fragment. The Auxiliary Data Block used for AUX_DATA_START is defined as:

```
typedef struct _AUX_DATA_START_BLOCK_
{
        AUX_DATA_BLOCK_HDR      hdr;
        VOID                    *auxDataPhysAddr;
        UINT32                  auxDataSize;
}AUX_DATA_START_BLOCK;
```

hdr.auxType	The auxType field of the Auxiliary Data Block is set to AUX_DATA_START.
hdr.auxVersion	Set to AUX_DATA_BLOCK_HDR_VERSION
hdr.auxLength	Set to sizeof(AUX_DATA_START_BLOCK) (24)
hdr.auxPayloadVersion	Set to AUX_DATA_START_VERSION
auxDataPhysAddr	This field contains the Physical Address for the beginning of Auxiliary Data, or zero. This field may be set to zero by protocol stacks prior to the delivery of the transmit ECB to the LSL. If the field is zero in the transmit path the LSL will fill the field with the Physical Address. The physical address is typically placed in this field by the code that creates the Auxiliary Data fragment. The Physical Address is typically used by actual hardware adapters. MLIDs and HSMs must fill this address with either the Physical Address, or zero.
auxDataSize	This field contains the total size of the auxiliary data space. The value stored in this field includes both used and unused space. This value can be used to calculate the amount of auxiliary data space that is not currently used for auxiliary data blocks. On transmits, this field will hold the same value that is stored in the FragmentLength field of the FRAGMENT_STRUCT associated with Auxiliary Data.

2.3.2 Auxiliary Data End

AUX_DATA_END is the last auxiliary data block in an Auxiliary Data fragment. The Auxiliary Data Block used for AUX_DATA_END is defined as follows:

```
typedef struct _AUX_DATA_END_BLOCK_
        AUX_DATA_BLOCK_HDR      hdr;
        UINT32                  terminator;
}AUX_DATA_END_BLOCK;
```

hdr.auxType	The auxType field of the Auxiliary Data Block is set to AUX_DATA_END.
hdr.auxVersion	Set to AUX_DATA_BLOCK_HDR_VERSION
hdr.auxLength	Set to sizeof(AUX_DATA_END_BLOCK) (20)
hdr.auxPayloadVersion	Set to AUX_DATA_END_VERSION

terminator This field is set to NULL.

2.3.3 Auxiliary Data NULL

AUX_DATA_NULL is an optional element. If present the AUX_DATA_NULL auxiliary data block must fall between AUX_DATA_START and AUX_DATA_END auxiliary data blocks. The AUX_DATA_NULL auxiliary block is a place-holder and is to be skipped over when scanning the auxiliary data buffer. The AUX_DATA_NULL auxiliary data block enables the deletion and insertion of auxiliary data blocks in the auxiliary data buffer without performing a copy. The auxiliary data block used for AUX_DATA_NULL is defined as follows:

```
typedef struct _AUX_DATA_NULL_BLOCK_
{
        AUX_DATA_BLOCK_HDR      hdr;
}AUX_DATA_NULL_BLOCK;
```

hdr.auxType	The auxType field of the Auxiliary Data Block is set to AUX_DATA_NULL.
hdr.auxVersion	Set to AUX_DATA_BLOCK_HDR_VERSION
hdr.auxLength	The size of the AUX_DATA_BLOCK_HDR plus any additional space associated with the NULL auxiliary data block.
hdr.auxPayloadVersion	Set to AUX_DATA_NULL_VERSION

3 ODI Structures

This section describes the changes required to other ODI structures to identify components that are ECB Auxiliary Data aware.

3.1 Protocol Stacks

3.1.1 Configuration Table

PConfigTable_ODISpecMajorVersion	This field contains the major version of the ODI Specification that the Protocol Stack is written to. For example, if the version of the ODI Specification is 1.11, the value of this field is 1. The ECB Extensions are being proposed for ODI Specification version 1.20. This field will remain at 1.
PConfigTable_ODISpecMinorVersion	This field contains the minor version of the ODI Specification that the Protocol Stack is written to. For example, if the version of the ODI Specification is 1.11, the value of this field is 11. The ECB Extensions are being proposed for ODI Specification version 1.20. This field will be set to 20.
PConfigTable_SystemFlags	Added a bit definition for indicating AUX data support:

PSTK_CFG_AUX_SUP_BIT	The protocol sets this bit if it is ECB auxiliary data aware. However, it does not imply that the protocol will interpret or act upon the information contained in the auxiliary data blocks.

3.1.2 Assembly Spec Stacks

Recognizing that on the NetWare 5.1 platform the protocols are currently written to the Assembly spec, this section details the changes needed to minimally indicate Aux Data support.

3.1.2.1 Configuration Table

CFGMajorVersion	This field contains the major version of the Config table structure definition. This field will remain at 1.
CFGMinorVersion	This field contains the minor version of the Config table structure definition. This field should be changed to 11.
Stack_Flags	A New field that contains the new bit definitions to indicate support for MP and AUX data support.
AUX_DATA_SUPPORT	The new bit defined to indicate support for Aux data.

3.2 MLIDs

3.2.1 MLIDCFG_Flags

MF_ECB_AUX_SUP_BIT	The MLID sets this bit if it is ECB auxiliary data aware. However, it does not imply that the MLID knows how to interpret or act upon the information contained in the auxiliary data blocks. It does mean that the MLID can safely process ECBs that contain auxiliary data. For example, if a transmit ECB with auxiliary data was handed to the MLID for processing, the MLID would not process the last fragment in the fragment list, treating it as if it were data to be transmitted.

3.3 C MSM

The C MSM configuration information returned by CMSMGetConfigInfo.

3.3.1 CMSM_CONFIG_TABLE

CMSMCFG_ODISpecMajorVersion	This field contains the major version of the ODI Specification that this version of the C MSM is written to. For example, if the version of the ODI Specification is 1.11, the value of this field is 1. For this proposal, this field will remain at 1.
CTSMCFG_ODISpecMinorVersion	This field contains the minor version of the ODI Specification that this version of the C MSM is written to. For example, if

the version of the ODI Specification is 1.11, the value of this field is 11. For this proposal this field will be set to 20.

3.4 C TSM

The <CTSM> configuration information returned by <CTSM>GetConfigInfo.

3.4.1 CTSM_CONFIG_TABLE

CTSMCFG_ODISpecMajorVersion — This field contains the major version of the ODI Specification that this version of the <CTSM> is written to. For example, if the version of the ODI Specification is 1.11, the value of this field is 1. For this proposal this field will remain at 1.

CTSMCFG_ODISpecMinorVersion — This field contains the minor version of the ODI Specification that this version of the <CTSM> is written to. For example, if the version of the ODI Specification is 1.11, the value of this field is 11. For this proposal this field will be set to 20.

3.5 C HSM

3.5.1 DRIVER_PARM_BLOCK

The HSMSpecVersionStringPtr field in the Driver Parameter Block will be updated to reflect the version of the ODI Specification.

HSMSpecVersionStringPtr — Pointer to the version string that describes the version of the HSM specification to which the HSM is written. For this proposal, the string is defined by Novell as “HSM_CSPEC_VERSION: 1.20”.

3.5.2 MLID_CONFIG_TABLE

MLIDCFG_SGCount — This field contains the maximum number of scatter/gather elements the adapter is capable of handling. The C HSM sets this variable. This field is only valid if the MM_FRAGS_PHYS_BIT bit in the MLIDCFG_ModeFlags field is set. The minimum value is 2 for non-ECB aware C HSMs (1 for the MAC header and 1 for data). The minimum value is 1 for ECB aware C HSMs. The maximum value is 17 (1 for the MAC header and 16 for data).

C HSMs that support auxiliary data need to adhere to the following guidelines for the minimum values. The minimum value is 3 for non-ECB aware (i.e., TCB aware) C HSMs. The minimum value is 2 for ECB aware C HSMs.

MLIDCFG_Flags

MF_ECB_AUX_SUP_BIT — The C HSM sets this bit if it is ECB auxiliary data aware. However, it does not imply that the C HSM knows how to interpret or act upon the information contained in the auxiliary data blocks. It does mean

that the C HSM can safely process ECBs that contain auxiliary data. For example, if a transmit ECB with auxiliary data was handed to the C HSM for processing, the C HSM would not process the last fragment in the fragment list, treating it as if it were data to be transmitted.

4 Impacts

This section describes the changes needed in ODI modules in order to support ECB Auxiliary Data.

4.1 C HSM

The HSM must ensure that the auxiliary data is handled correctly. The HSM will parse for auxiliary data blocks. When it finds an auxiliary data block where action is needed, it will perform the necessary processing based on the content of the auxiliary data block.

- Support Structure updates as described in Sections 2 and 3.
- Send and receive paths must be able to handle properly the presence of auxiliary data.

4.2 Protocol Stacks

Protocol Stacks must ensure that ECBs constructed or processed by the stack handle auxiliary data correctly. This applies to all types of stacks (i.e., bound, prescan, and default). Below are some of the areas that protocol stacks will need to update:

- Support structure updates as described in Sections 2 and 3.
- The data path must be auxiliary data aware.
- Send and receive paths must be able to handle properly the presence of auxiliary data.

4.3 Other NLMs

Some NLMs have chosen to create their own transmit ECBs. These NLMs that are ECB aware must ensure that ECBs constructed or processed by an NLM handle auxiliary data correctly. The things that these NLMs must be aware of include the following:

- Must support structure updates as described in Sections 2 and 3.
- Must verify that data path is auxiliary data aware.
- Send and receive paths must be able to handle properly the presence of auxiliary data.

4.4 ODI.H

This section summarizes the modifications to odi.h. Comments are not spelled out in this section but can be gathered from the previous section where they are described in detail.

The minor version define in the C ODI Specification Version Numbers section has been changed to the following:

#define ODI_SPEC_MINOR_VER 20

Added the following to the PConfigTable_SystemFlags definition:

#define PSTK_CFG_AUX_SUP_BIT 0x00000002

Added the following to Stack ID Definitions section:

Raw Sends

#define	TX_RAW_SEND_PRIORITY_0	0xFF	0 = No Priority
#define	TX_RAW_SEND_PRIORITY_1	0xFE	1 = Lowest Priority
#define	TX_RAW_SEND_PRIORITY_2	0xFD	
#define	TX_RAW_SEND_PRIORITY_3	0xFC	
#define	TX_RAW_SEND_PRIORITY_4	0xFB	
#define	TX_RAW_SEND_PRIORITY_5	0xFA	
#define	TX_RAW_SEND_PRIORITY_6	0xF9	
#define	TX_RAW_SEND_PRIORITY_7	0xF8	7 = Highest Priority

Normal Sends

#define	TX_SEND_PRIORITY_0	0xF7	0 = No Priority
#define	TX_SEND_PRIORITY_1	0xF6	1 = Lowest Priority
#define	TX_SEND_PRIORITY_2	0xF5	
#define	TX_SEND_PRIORITY_3	0xF4	
#define	TX_SEND_PRIORITY_4	0xF3	
#define	TX_SEND_PRIORITY_5	0xF2	
#define	TX_SEND_PRIORITY_6	0xF1	
#define	TX_SEND_PRIORITY_7	0xF0	7 = Highest Priority

/* 0xFFFy Values from 0xFFF0 – 0xFFFF (zn = 0xFF) indicate legacy raw and normal send definitions for the ECB_StackID.

0xzn The **z** and **n** nibbles of the zn byte provide additional information about the transmit request. The z and n nibbles are defined as follows:

0xz The **z** nibble of the **zn** byte has a pattern of 10xx. The following values are defined for the z: */

#define	TX_NO_CHECKSUM	0x8	No Checksum generation on this packet.
#define	TX_TRANSPORT_CHECKSUM	0x9	Generate the Transport Layer Checksum
#define	TX_NETWORK_CHECKSUM	0xA	Generate the Network Layer Checksum (Ipv4)

/* 0xn The **n** nibble of the zn byte indicates a value of 0x0 – 0xF. The following values are defined for the n nibble of the zn byte: */

```
        #define TX_AUX_DATA 0x1                 /* Auxiliary Data is present. */
```

Added the following to the Rx Packet Attributes section:

```
#define  PA_RX_AUX_DATA      0x00800000      /* Set if auxiliary data is present. */
```

Added the following to the MLIDCFG_FLAGS bit definition section:

```
#define  MF_ECB_AUX_SUP_BIT     0x2000h
```

Created a new section in the area of the ECB definition and added the following:

```
#define  AUX_DATA_BLOCK_HDR_VERSION    1

/* Auxiliary Data Block Types */

#define  AUX_DATA_START                1
#define  AUX_DATA_END                  2
#define  AUX_DATA_NULL                 3

typedef struct _AUX_DATA_BLOCK_HDR_
{
        UINT32          auxType;
        UINT32          auxVersion;
        UINT32          auxLength;
        UINT32          auxPayloadVersion;
}AUX_DATA_BLOCK_HDR;

#define  AUX_DATA_START_VERSION        1

typedef struct _AUX_DATA_START_BLOCK_
{
        AUX_DATA_BLOCK_HDR      hdr;
        VOID                    *auxDataPhysAddr;
        UINT32                  auxDataSize;
}AUX_DATA_START_BLOCK;

#define  AUX_DATA_END_VERSION          1

typedef struct _AUX_DATA_END_BLOCK_
{
        AUX_DATA_BLOCK_HDR      hdr;
        UINT32                  terminator;
}AUX_DATA_END_BLOCK;

#define  AUX_DATA_NULL_VERSION         1

typedef struct _AUX_DATA_NULL_BLOCK_
{
        AUX_DATA_BLOCK_HDR      hdr;
}AUX_DATA_NULL_BLOCK;
```

5 LSL APIs

This section defines APIs provided by the LSL for use by protocol stacks and MLIDs.

5.1 LSLAllocateAuxDataBuffer

Allocates and initializes an auxiliary data buffer. Generally used by protocol stacks.

Syntax
```
long    LSLAllocateAuxDataBuffer (
                long size,
                RTag *rTag,
                long blkSize,
                void **bufferPtr,
                void **blkPtr ) ;
```

Input Parameters

size	Minimum size required for the auxiliary data buffer.
rTag	Pointer to a valid resource tag with signature of *ECBSignature*.
blkSize	Size of NULL block to be added to the buffer between the START and END blocks if a NULL block is to be created. If a NULL block is not to be created then this field is set to 0.
bufferPtr	Destination where the pointer to the new auxiliary data buffer is to be stored.
blkPtr	Destination to store the pointer to the NULL block created within the auxiliary data buffer if one was requested. Set to NULL if *blkSize* is zero or if the location within the auxiliary data block is not required.

Output Parameters

bufferPtr	Contains the pointer to where the new auxiliary data buffer is to be stored.
blkPtr	Contains the pointer to the NULL block created within the auxiliary data buffer if one was requested.

Return Values

Successful	The requested operation was completed successfully.
BadParameters	One or move input parameters were invalid.
OutOfResources	Resources were not available to fill the request.

Remarks

Allocates an auxiliary data buffer with valid START and END blocks and an optional NULL block. The size of the buffer and the NULL block are specified at call time. The optional NULL block can be used as a space holder for another block type that will overwrite the NULL block later.

5.2 LSLReturnAuxDataBuffer

Returns aux data buffers back to the system. Generally used by protocol stacks.

Syntax long LSLReturnAuxDataBuffer (
void *bufferPtr) ;

Input Parameters

bufferPtr	Pointer to the auxiliary data buffer being returned.

Output Parameters

None

Return Values

Successful	The requested operation was completed successfully.
BadParameters	The parameter was invalid.

Remarks

Returns an aux data buffer acquired with *LSLAllocateAuxDataBuffer*.

5.3 LSLInitializeAuxDataBuffer

Initializes an auxiliary data buffer by formatting the START and END auxiliary data blocks. Optionally inserts a NULL auxiliary data block if specified by the caller. Generally used by protocol stacks and MLIDs.

Syntax long LSLInitializeAuxDataBuffer (
long size,
void *bufferPtr,
long blkSize,
void **blkPtr) ;

Input Parameters

size	The size of the auxiliary data buffer in bytes.
bufferPtr	Pointer to an existing auxiliary data buffer.
blkSize	The size of NULL auxiliary data block to be inserted between START and END auxiliary data blocks. If a NULL auxiliary data block is not to be inserted, set this field to zero.
blkPtr	Destination where the NULL auxiliary data block pointer is to be stored. Set to NULL if *blkSize* is zero or if the location within the auxiliary data block is not required.

Output Parameters

blkPtr	Contains a pointer to the NULL auxiliary data block.

Return Values

Successful	The requested operation was completed successfully.
BadParameters	One or more parameters were invalid.

Remarks

Initializes an auxiliary data buffer with the START and END auxiliary blocks with an optional NULL auxiliary data block between them.

5.4 LSLInsertBlockIntoAuxDataBuffer

Inserts a new NULL auxiliary data block into an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax
```
long    LSLInsertBlockIntoAuxDataBuffer (
            void *bufferPtr,
            long size,
            void **blkPtr ) ;
```

Input Parameters

bufferPtr	Pointer to a valid auxiliary data buffer.
size	The size of the NULL block to be inserted.
blkPtr	The destination where the pointer to the NULL auxiliary data block will be stored. Set to NULL if the location of the block within the auxiliary data buffer is not required.

Output Parameters

blkPtr	Contains a pointer to the NULL auxiliary data block.

Return Values

Successful	The requested operation was completed successfully.
BadParameters	One or more of the parameters were invalid.
OutOfResources	Resources were not available to fill the request.

Remarks

Finds a NULL auxiliary block of the correct size or breaks a larger NULL auxiliary data block into two NULL auxiliary data blocks, one being the requested size. If there is not an existing NULL auxiliary data block that meets the requirement then a new NULL auxiliary block of the requested size is created in front of the END block.

5.5 LSLRemoveBlockFromAuxDataBuffer

Removes the specified auxiliary data block from an auxiliary data buffer by reformatting it into a NULL block. Generally used by protocol stacks and MLIDs

Syntax long LSLRemoveBlockFromAuxDataBuffer (
void *bufferPtr,
void *blkPtr) ;

Input Parameters

bufferPtr	Pointer to the auxiliary data buffer.
blkPtr	Pointer to the auxiliary data block to be removed.

Output Parameters

None

Return Values

Successful	The requested operation was completed successfully.
BadParameters	One or more of the parameters were invalid.

Remarks

An existing auxiliary data block is removed by being converted into a NULL auxiliary data block.

5.6 LSLFindBlockInAuxDataBuffer

Locates a specified auxiliary data block in an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax long LSLFindBlockInAuxDataBuffer(
void *bufferPtr,
long blockType,
void **blockPtr) ;

Input Parameters

bufferPtr	Pointer to the beginning of a valid auxiliary data buffer or a pointer to a valid auxiliary data block within an auxiliary data buffer.
blockType	Block type to search for.
blockPtr	Destination to store pointer of requested auxiliary data block. Set to NULL if caller wants to test if type exists but does not care where it is located within the auxiliary data buffer.

Output Parameters

blockPtr	Contains the pointer to requested auxiliary data block.

Return Values

Successful	The requested operation was completed successfully.
BadParameters	One or more of the parameters were invalid.
ItemNotPresent	An auxiliary data block of the requested type was not found.

Remarks

Locates an auxiliary data block of requested type within an auxiliary data buffer. The *bufferPtr* does not need to point to the START auxiliary data block if used as a find next auxiliary block. If used as a find next, the parameter *bufferPtr* contains the address of a valid auxiliary data block to begin the search.

Input parameter *blockPtr* can be set to NULL if caller wants to test if type exists but does not care where it is located within the auxiliary data buffer.

5.7 LSLCompressAuxDataBuffer

Removes all NULL auxiliary data blocks from an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax
```
long    LSLCompressAuxDataBuffer (
            void *bufferPtr ) ;
```

Input Parameters

bufferPtr	A pointer to a valid auxiliary data buffer.

Output Parameters

None

Return Values

Successful	The requested operation was completed successfully.
BadParameters	The input parameter was invalid.

Remarks

Removes all NULL auxiliary data blocks from an auxiliary data buffer.

5.8 LSLExpandAuxDataBuffer

Inserts a NULL auxiliary data block in front of the END auxiliary data block. Generally used by protocol stacks and MLIDs

Syntax
```
long    LSLExpandAuxDataBuffer (
            void *bufferPtr ) ;
```

Input Parameters

bufferPtr	A pointer to a valid auxiliary data buffer.

Output Parameters

None

Return Values

Successful	The requested operation was completed successfully.
BadParameters	The input parameter was invalid.
OutOfResources	Resources were not available to fill the request.

Remarks

Expands the auxiliary data buffer by adding a NULL auxiliary data block before the END auxiliary block equal to the size of all the available space in the auxiliary data buffer. All other NULL auxiliary blocks are decompressed so that the auxiliary data buffer will contain only one NULL auxiliary data block of the maximum size possible.

5.9 LSLValidateAuxDataBuffer

Runs through the auxiliary data buffer and checks for consistency of the auxiliary data blocks. Generally used by protocol stacks and MLIDs

Syntax long LSLValidateAuxDataBuffer (
void *bufferPtr) ;

Input Parameters

bufferPtr	Pointer to the auxiliary data buffer to validate.

Output Parameters

None

Return Values

Successful	The requested operation was completed successfully.
BadParameters	Pointer to the auxiliary data buffer is invalid.
BadCommand	The auxiliary data buffer pointed to by *bufferPtr* is invalid.

Remarks

Used to test an auxiliary data buffer to see if it is well formed.

6 C MSM APIs

This section defines APIs provided by the C MSM for use by C HSMs.

6.1 CMSMInitializeAuxDataBuffer

Initializes an auxiliary data buffer by formatting the START and END auxiliary data blocks. Optionally inserts a NULL auxiliary data block if specified by the caller. Generally used by protocol stacks and MLIDs.

Syntax ODISTAT CMSMInitializeAuxDataBuffer (
UINT32 size,
void *bufferPtr,
UINT32 blkSize,
void **blkPtr) ;

Input Parameters

size	The size of auxiliary data buffer in bytes.
bufferPtr	Pointer to an existing auxiliary data buffer.
blkSize	The size of NULL auxiliary data block to be inserted between START and END auxiliary data blocks. If a NULL auxiliary data block is not to be inserted, set this field to zero.
blkPtr	Destination where the NULL auxiliary data block pointer is to be stored. Set to NULL if *blkSize* is zero or if the location within the auxiliary data block is not required.

Output Parameters

blkPtr	Contains a pointer to the NULL auxiliary data block.

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	One or more parameters were invalid.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

Initializes an auxiliary data buffer with the START and END auxiliary blocks with an optional NULL auxiliary data block between them.

6.2 CMSMInsertBlockIntoAuxDataBuffer

Inserts a new NULL auxiliary data block into an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax ODISTAT CMSMInsertBlockIntoAuxDataBuffer (
void *bufferPtr,
UINT32 size,
void **blkPtr) ;

Input Parameters

bufferPtr	Pointer to a valid auxiliary data buffer.
size	The size of the NULL block to be inserted.
blkPtr	The destination where the pointer to the NULL auxiliary data block will be stored. Set to NULL if the location of the block within the auxiliary data buffer is not required.

Output Parameters

blkPtr	Contains a pointer to the NULL auxiliary data block.

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	One or more of the parameters were invalid.
ODISTAT_OUT_OF_RESOURCES	Resources were not available to fill the request.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

Finds a NULL auxiliary block of the correct size or breaks a larger NULL auxiliary data block into two NULL auxiliary data blocks, one being the requested size. If there is not an existing NULL auxiliary data block that meets the requirement then a new NULL auxiliary block of the requested size is created in front of the END block.

6.3 CMSMRemoveBlockFromAuxDataBuffer

Removes the specified auxiliary data block from an auxiliary data buffer by reformatting it into a NULL block. Generally used by protocol stacks and MLIDs

Syntax ODISTAT CMSMRemoveBlockFromAuxDataBuffer (
void *bufferPtr,
void *blkPtr) ;

Input Parameters

bufferPtr	Pointer to the auxiliary data buffer.
blkPtr	Pointer to the auxiliary data block to be removed.

Output Parameters

None

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	One or more of the parameters were invalid.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

An existing auxiliary data block is removed by being converted into a NULL auxiliary data block.

6.4 CMSMFindBlockInAuxDataBuffer

Locates a specified auxiliary data block in an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax
```
ODISTAT     CMSMFindBlockInAuxDataBuffer(
                void *bufferPtr,
                UINT32 blockType,
                void **blockPtr ) ;
```

Input Parameters

bufferPtr	Pointer to the beginning of a valid auxiliary data buffer or a pointer to a valid auxiliary data block within an auxiliary data buffer.
blockType	Block type to search for.
blockPtr	Destination to store pointer of requested auxiliary data block. Set to NULL if caller wants to test if type exists but does not care where it is located within the auxiliary data buffer

Output Parameters

blockPtr	Contains the pointer to the requested auxiliary data block.

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	One or more of the parameters were invalid.
ODISTAT_ITEM_NOT_PRESENT	An auxiliary data block of the requested type was not found.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

Locates an auxiliary data block of requested type within an auxiliary data buffer. The *bufferPtr* does not need to point to the START auxiliary data block if used as a find next auxiliary block. If used as a find next, the parameter *bufferPtr* contains the address of a valid auxiliary data block to begin the search.

Input parameter *blockPtr* can be set to NULL if caller wants to test if type exists but does not care where it is located within the auxiliary data buffer.

6.5 CMSMCompressAuxDataBuffer

Removes all NULL auxiliary data blocks from an auxiliary data buffer. Generally used by protocol stacks and MLIDs

Syntax ODISTAT CMSMCompressAuxDataBuffer (
void *bufferPtr) ;

Input Parameters

bufferPtr A pointer to a valid auxiliary data buffer.

Output Parameters

None

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	The input parameter was invalid.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

Removes all NULL auxiliary data blocks from an auxiliary data buffer.

6.6 CMSMExpandAuxDataBuffer

Inserts a NULL auxiliary data block in front of the END auxiliary data block. Generally used by protocol stacks and MLIDs

Syntax ODISTAT CMSMExpandAuxDataBuffer (
void *bufferPtr) ;

Input Parameters

bufferPtr A pointer to a valid auxiliary data buffer.

Output Parameters

None

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	The input parameter was invalid.
ODISTAT_OUT_OF_RESOURCES	Resources were not available to fill the request.
ODISTAT_BAD_COMMAND	This API is not supported on this version of NetWare.

Remarks

Expands the auxiliary data buffer by adding a NULL auxiliary data block before the END auxiliary block equal to the size of all the available space in the auxiliary data buffer. All other NULL auxiliary blocks are decompressed so that the auxiliary data buffer will contain only one NULL auxiliary data block of the maximum size possible.

6.7 CMSMValidateAuxDataBuffer

Runs through the auxiliary data buffer and checks for consistency of the auxiliary data blocks. Generally used by protocol stacks and MLIDs

Syntax
```
ODISTAT     CMSMValidateAuxDataBuffer (
                void *bufferPtr ) ;
```

Input Parameters

bufferPtr	Pointer to the auxiliary data buffer to validate.

Output Parameters

None

Return Values

ODISTAT_SUCCESSFUL	The requested operation was completed successfully.
ODISTAT_BAD_PARAMETER	Pointer to the auxiliary data buffer is invalid.
ODISTAT_BAD_COMMAND	The auxiliary data buffer pointed to by *bufferPtr* is invalid or this API is not supported on this version of NetWare.

Remarks

Used to test an auxiliary data buffer to see if it is well formed.

APPENDIX A

	ECB Extension Aware Protocol Stack	Existing Protocol Stack (non-aware)
ECB Extension Aware MLID/HSM	Bind Allowed	Bind NOT Allowed
Existing MLID/HSM (non-aware)	Bind Allowed	Bind Allowed

Table: 1 Bindings Allowed with ECB Extensions.

Issues with Binding:

1. As HSMs or MLIDs do not know if the packet received will be handed to a Protocol Stack that is unaware of the ECB extensions, the binding of such a combination is precluded.

2. Protocol Stacks can cause MLIDs/HSMs to start to pass auxiliary data. They are also aware of non-extension formats, so new Aux-Aware Protocol Stacks can bind to older/Non-Auxiliary data MLIDs/HSMs.