Interface Specification

DMPP-2020

Datamaxx Message Processing Protocol®
(DMPP-2020)
DMPP-2020® Interface Specification
Version 9.0

Datamaxx®
APPLIED TECHNOLOGIES

www.Datamaxx.com

Published by:

Datamaxx Group, Inc. d/b/a
Datamaxx Applied Technologies, Inc.
2001 Drayton Drive
Tallahassee, FL 32311-7854
(850) 558-8000
www.Datamaxx.com

Revision History:

Version	Date	Notes
Version 0	Aug 1996	1) Original Issue
Version 1	Oct 1996	1) Added Levels of Service 2) Changed references to "Sequence" numbers to "validation" numbers to avoid confusion with application message sequence numbers 3) Edited in order to clarify some portions, without changing meanings or specification 4) Added Trademark notices and formal specification name
Revision 2	Jun 1997	1) Clarified use of Network Byte Order in all structures, and defined the Start and Stop Patterns 2) Minor editorial and grammatical changes
Revision 3	Sept 1997	1) Registered Copyright – TX 4-624-223 2) Modified references to NCIC 3) Changed block length field to positive value 4) Added a maximum buffer size specification. Changed format specification for the validation field.
Revision 4	Sept 1997	1) Clarified block structure and length field definitions
Revision 5	Sept 1998	1) Editorial changes defining that the implementer must choose the encryption algorithm 2) Editorial changes to the maximum overall data length field 3) Updated Datamaxx address and telephone number
Revision 6	Jun 2000	1) Added ability to send messages in multiple blocks: increased maximum block length and data length; and added new function codes 2) Redefined original concept of "Status Codes" to "Status Codes for Request Messages". The latter allows an indication of whether or not a message contains a binary object, and if so, the type.
Revision 7	Feb 2004	Updated format
Revision 8	Feb 2005	Added support for 256-bit AES encryption
Revision 8.1	Apr 2005	Added clarification regarding 256-bit AES encryption
Revision 9.0	Mar 2015	Added Encryption Types 4 & 5, the XML Key book format, and miscellaneous clarifications.

TABLE OF CONTENTS

1.0 INTRODUCTION

The purpose of this paper is to define a specification that can be implemented to provide robust message handling in the Law Enforcement Environment. As the transition to modern communications protocols continues, new problems and challenges are presented to developers. This is especially true with “Open Systems”, in which there are components from various vendors, all of which must operate in harmony.

With legacy systems, one vendor had control of processing, from the end-user keyboard to the host system, and thus could control all standards, and could implement necessary functionality to ensure that all messages were delivered reliably.

With Open Systems and diverse vendors, functionality tends to be implemented as a series of layers or services, with information being passed up and down between layers. Complicating this is the fact that the layers may be implemented as a series of disparate free running processes, in which data is passed back and forth between them. Therefore, an application may send data through several layers and processes about which it has no knowledge. Each process or layer may acknowledge to the previous process or layer that the data was successfully processed; however, error messages are often not communicated to previous processes, and the originating application may not be aware of an error. Therefore, there is a need for “application-to-application” or “end-to-end” acknowledgment.

Complicating the situation is that Open Systems are truly open, as they are designed to allow easy interconnection. This immediately provides points of access that can be used for unauthorized or abusive use of a system.

A further factor is that many protocols are “peer-to-peer” and do not provide a continuous status monitoring (as is the case with “master slave” type protocols). This can lead to situations in which an application can send a message to a destination that cannot process it. Since there is no immediately available status, error indications may not be provided for several minutes (or at all) and the sending application will not be aware of the situation.

Consider the following scenario:

1) Host prepares a message for transmission.

2) Host passes the message to communications subsystem.

3) Communications subsystem passes message to communications controller.

4) Subsystem sends message immediately to destination, but is not aware if any intermediate devices (e.g. bridges or routers) are inoperative.

5) Remote application crashes before reading buffer, or operator powers system off.

In this scenario, the host application would consider that the messages have been correctly processed, when indeed it was not. Furthermore, many messages may have been sent and buffered for a remote application that never processes them. There are also many other potential points of failure that can leave the host in a state assuming a message was delivered, when it was not actually delivered.

In order to eliminate these potential points of failure, a structure must be defined that can be used universally to guarantee message delivery between two endpoints, regardless of the intermediate processes, services, etc. that it traverses.

The approach defined herein uses a Message Header processing to achieve full end-to-end confirmation of all messages.

The processing strategy is known as the "Datamaxx Message Processing Protocol (DMPP-2020®[1])".

[1] Datamaxx Message Processing Protocol and DMPP-2020 are registered trademarks of Datamaxx Applied Technologies, Inc.

2.0 CONCEPTS

In developing the message header processing, many factors were considered including:

- Compatibility with NCIC designs
- Compatibility with State designs
- Full message delivery confirmation
- Communications Protocol Independent
- Applicable to all processing platforms
- Programmer friendly
- Support for security issues
- Support for data encryption
- Features can be configured to meet different requirements
- Flow control is automatically provided to avoid flooding of a target system

The design that evolved, after much research, involves the implementation of a special header in each message packet. This header contains control fields that can be used to provide all functionality, as needed. This header is referred to as the "**Extended Message Header**" throughout this specification.

A discussion of each of the concepts is warranted, in order to provide background and rationale for the design.

1) **Compatibility with NCIC designs** – This design is compatible with current and future NCIC designs (e.g. NCIC-2000/NCIC-2000 3rd generation).

2) **Compatibility with State designs** – This design allows the Extended Message Header to be placed in front of existing message formats, thus alleviating the need to modify existing processing applications.

3) **Full Message Delivery Confirmation** – The Extended Message Header provides both *positive and negative* confirmation of message delivery. For negative delivery confirmation, a reason code is provided.

4) **Communications Protocol Independent** – Although the obvious protocol that this specification can be applied to is TCP/IP, it is actually protocol independent. It can operate on *any* binary transparent protocol, ranging from serial links to mainframe protocols (e.g. LU 6.2).

5) **Applicable to all Processing Platforms** – This design is compatible with *all* processing platforms. We paid careful attention to the sizing and alignment of all data fields in order to avoid alignment and size specification errors that are generated by some processors.

6) **Programmer Friendly** – The design guards against assumptions made by various compilers. For example, some compilers will automatically initialize data structures to null values, *or just plain junk*. This can lead to subtle processing flaws. Thus, this specification does not allow any command, directive or response code that is all null values, and requires that all values be verified. It is also programming language independent. All Extended Message Header processing is symmetrical with respect to direction (inbound and outbound).

7) **Support for Security Issues** – The Extended Message Header provides for full authentication of all connections, including dynamic re-verification of connections at random intervals (see "Coded Messages" below).

8) **Support for Data Encryption** – The Extended Message Header provides for full encryption of the data portion of messages. This allows a full software solution to be implemented, independent of all communications hardware. Dynamic key update and control is supported. Encryption keys can be dynamically selected from the "key book" (explained below), so that the same key is not used over and over again during encrypted message exchanges.

9) **Features can be configured to Meet Different Requirements** – The features can be configured to meet the needs of a specific system. For example, the Extended Message Header can be implemented using a few of its capabilities, and then more features can be activated as required.

10) **Levels of Implementation** – The specification can be implemented as "levels of service", depending on what options are selected enabling it to be adapted to many different needs and environments.

11) **Flow Control** – The Extended Message Header can provide a natural flow control, if desired by the implementer. For example, the sender of the message can choose to send the message with ACK, and will then wait until the receiver sends the ACK before proceeding, thus allowing the sender to pace at which the receiver can operate.

3.0 EXTENDED MESSAGE HEADER

The Extended Message Header is a structure that is inserted in a cleanly delineated message block. The general structure of the message block is detailed in the table shown below.

FIELD	SPECIFICATION
Start Pattern (STAP)	\xFF\x00\xAA\x55
Block Length	32-bit singed integer (see note below) Encompasses the whole packet, including the Start Pattern, Block Length field itself, Extended Message Header, Data (if any present) and Stop Pattern
Header	Extended Message Header (defined in Section 4)
Data	Variable length data
Stop Pattern (STOP)	\x55\xAA\x00\xFF *(the reverse of the Start Pattern)*

For consistency across platforms, all values in the header are stored in “Network Byte Order”. This order places the most significant byte first, descending to the least significant byte reading to the right.

The minimum block size is 28 characters, which can occur when the Extended Message Header length is 16 and there is no data present. The maximum block size is 2,147,483,647 ($2^{31} – 1$). Therefore, the value of the Block Length field must never be less than 28 or more than 2,147,483,647.

4.0 EXTENDED MESSAGE HEADER FORMAT

The Extended Message Header has the following ***required*** format:

FIELD	SPECIFICATION
Header Length	16-bit signed integer
Function Code	16-bit signed integer
Validation Code	32-bit unsigned integer
Data Length	32-bit signed integer
Status Code	16-bit signed integer
Destination	16-bit signed integer

The Extended Message Header has the following **optional** extension for encryption:

FIELD	SPECIFICATION
Encryption Header Length	16-bit signed integer
Encryption Type	16-bit signed integer
Parameters	Variable, depending on Encryption Type as defined in below

In the following tables, all numbers are expressed as integers. They can be converted to other number systems (e.g. octal or hex) as required. *Note how the use of zeros is consistently avoided*. Each field is discussed in detail, as follows:

1) **Header Length** encompasses all the header data, including the length field. It will be 16 if an Encryption Header is not included. If an Encryption Header is included, Header Length will be at least 20 (*the actual value depends on the type of encryption specified in the header*).

2) **Function Code** defines the processing path of the message. The following is the list of supported function codes:

VALUE	DESCRIPTION
1	Data message with no acknowledgment, final block
2	Data message with acknowledgment, final block
3	Data message with no acknowledgment, more blocks to follow (see note below)
4	Data message with acknowledgment, more blocks to follow (see note below)
17	Positive acknowledgment to data message (Status Code is set to “Successful receipt of data message”)
18	Negative acknowledgment to data message (Error is defined in the Status Code field)
33	Request status of system
34	Response to status request (Status is defined in the Status Code field)

VALUE	DESCRIPTION
49	Send Coded Message 1
50	Send Coded Message 2
65	Positive response to Coded Message 1
66	Positive response to Coded Message 2

Note: *Function Codes 3 and 4 are used to indicate that the message will be sent in multiple blocks with Function Codes 1 and 2 used to indicate the last block. Each block in such messages must use successive values as the Validation Code.*

3) **Validation Code** defines a number that is used to create a unique identification for each message, and will be returned on its corresponding acknowledgment. Its format is up to the implementer. This value is not inspected but simply returned to the requester intact.

4) **Data Length** defines the length of the actual data portion of the message. It is used for redundancy checking. *It must be zero for status and status response messages*. The maximum value is 2,147,483,619 ($2^{31} - 1 - 28$).

5) **Status Code for Request Messages** contains the status code that can be included in request messages. The following is the list of supported status codes for Request Messages:

VALUE	DESCRIPTION
1	Message does not contain binary object
2	Message contains binary object in NCIC transaction format
3	Message contains binary object in NCIC response format
4	Message contains binary object in DSEO-2020[2] format

Note: *Any message that can contain a binary object in any of the supported formats can contain multiple binary objects but they must all be in the same format*.

6) **Status Code for Response Messages** contains the status code that can be returned in responses. They should be used only with responses – never part of request messages (i.e., status codes are not "piggybacked" onto a request). The code returned will depend on the type of request received (e.g. a write request with acknowledgment, or an explicit request for status). The following is the list of supported status codes for Response Messages:

VALUE	DESCRIPTION
1	Successful receipt of data message
17	Permanent (i.e., non-recoverable) error occurred (e.g. disk failure)
18	Temporary (i.e., recoverable) error occurred (e.g. printer out of paper)

[2] DSEO-2020 is a registered trademark of Datamaxx Applied Technologies, Inc.

VALUE	DESCRIPTION
19	Logical error occurred (e.g. too many messages received too quickly, and thus a queue containing acknowledgments filled up)
20	Message length exceeds maximum, message will be discarded
33	Queried destination is available and ready
34	Queried destination is available, but not ready (e.g. printer has buffer space, but is out of paper)
35	Queried destination is not available and not ready
49	Invalid function code received
50	Invalid (or non-existent) destination received
51	Invalid Extended Message Header format or length received
52	Function not supported

7) **Destination** defines a logical destination. This permits a packet to be addressed to different logical units, and effectively creates a cluster at a location. The actual definition is up to the implementer and the configuration. This permits logical units to be defined for specific purposes (e.g. a destination for "Hit Confirmation" messages), and permits implementation of message priorities. *A value of 0 is invalid*. The value of "-1" is considered a broadcast to all defined destinations.

8) **Encryption Header Length** defines the length of the optional encryption header. *A length of zero is invalid*. If this optional value is included, it will be at least four (4) (*the actual value depends on the type of encryption specified in the header as defined below*).

9) **Encryption Header** defines the parameters to be used to decrypt the data. The encrypted data itself is included in the Data field of the message. The format depends on the Encryption Type field as defined below. Note that the Encryption Header can specify that the Data field in the message is not encrypted.

FIELD	SPECIFICATION
Header Length	16-bit signed integer
	16-bit signed integer Defines how data is encrypted: 0 – No encryption 1 – 128-bit FIPS-197, CBC mode, PKCS7 padding 2 – 256-bit FIPS-197, CBC mode, PKCS7 padding 4 – 128-bit FIPS 140-2, CBC mode, PKCS7 padding 5 – 256-bit FIPS 140-2, CBC mode, PKCS7 padding ****Notice** Types 1 and 2 are included for backward compatibility, but systems that use those should change to use 4 or 5. A future version will likely deprecate support for Types 1 & 2.**

FIELD	SPECIFICATION
Parameters	Variable length depending on Encryption Type as defined below Specifies the parameters needed to decrypt the Data field in the message

a) **Encryption Type 0** – The contents of the Data field are not encrypted and, consequently, there are no associated parameters.

b) **Encryption Type 1** – The contents of the Data field are encrypted using parameters defined below with the National Institute of Standards and Technology (NIST) Advanced Encryption Standard (AES) as defined in the Federal Information Processing Standard (FIPS) 197, with the following options:

1) 128-bit keys

2) 128-bit encryption blocks

3) Cipher Block Chaining (CBC) Mode with an explicit Initialization Vector (IV)

4) PKCS7 padding

5) The encryption key should be derived from the book of keys identified by Book ID, using the specific key identified by Key ID. See Appendix A for the format of key books.

PARAMETER	SPECIFICATION
Book ID	16-bit signed integer (2 bytes)
Key ID	16-bit signed integer (2 bytes)
CRC	16-bit signed integer (2 bytes) Standard cyclic redundancy check (CRC-16) value of the clear-text with initial value set to zero
IV	16-bytes Initialization Vector *randomly* selected for each message

c) **Encryption Type 2** – Same as Type 1 with 256-bit keys

d) **Encryption Type 4** – Same as Type 1, except the encryption used is FIPS 140-2 compliant.

e) **Encryption Type 5** – Same as Type 1 with 256-bit keys, and the encryption used is FIPS 140-2 compliant.

f) The Encryption Header *should only be used* with Function Codes 1, 2, 3 and 4.

g) If DMPP-2020 blocking (Function Code of 3 or 4) is used, each block will be independently encrypted and decrypted using the CRC and IV included *in that block*.

h) The following errors will be NAK'd in the same manner as non-encrypted errors using the indicated status codes:

VALUE	DESCRIPTION
65	Invalid encryption header
66	Invalid book ID
67	Invalid key ID
68	CRC error

5.0 SERVICE LEVELS

The DMPP-2020 specification allows for service levels. A service level defines that functionality that has been activated for a given endpoint on a communications network. The following service levels are defined:

1) **Level 1** provides the functionality for handling message header functions from 1 through 47 (as they may be defined). This functionality encompasses guaranteed delivery of messages and full status checking, *but does not include authentication or encryption*.

2) **Level 2** provides the functionality as described in Level 1 and adds the functionality for system authentication (function codes 49 through 79).

3) **Level 3** provides the functionality as described in Level 2 plus adds the encryption options via the extensions for encryption.

6.0 IMPLEMENTATION NOTES

The following notes are presented to give an insight into how the Extended Message Header may be applied to various functions.

1) **Integer Values.** In this specification all integers are positive signed values, unless otherwise noted.

2) **Destination** does not have to replace existing header structures. It is meant to augment them. This technique permits many logical units to be addressed by a single host address (e.g. a single TCP/IP address).

3) **Flow Control.** By use of the "Write with Acknowledge" function, flow control may be achieved. The application can be structured to allow any number of messages to be outstanding at any time, subject only to the limits of the receiver. If the limit is set to 1, automatic flow control is achieved.

4) **Keep Alive Timer** provides full keep-alive support, at the application level. A keep-alive probe is a packet with a Request Status Function code and no data length. If an appropriate Response to Status Request is returned, then the connection is intact. Note that this can also be used to temporarily suspend traffic by responding with a Status Code 34 (temporarily unavailable).

5) **Coded Messages** are used to authenticate connections. Their use is specific, as follows:

    a) A session requesting a connection provides a predictive string of data (e.g. a logical name) and encodes it in such a way that the receiver can decode it. This can be done by using a known element (e.g. system name, date, circuit number, telephone number, etc.) and encoding it using a Huffman coding, Zip, SHA-2, or other encoding process. It sends it as Coded Message 1 to the receiver.

    b) The receiving session encodes a similar string (that is why it must be predictive) and compares it to the received string. If a match is found, a response code of 65 is sent, with no data. If no match, the receiver is silent (*Why tell the hacker how he or she failed?*).

    c) Either side of the session may send a Coded Message 2 request at any time. The Coded Message 2 has a random data string as its data portion. The receiver then adds another predictive string of data to the coded data, re-encodes it and returns it as a response of code 66 to the sender.

    d) The sender of the Coded Message 2 analyzes the response. If valid, processing continues (there is no response). If invalid, the connection is terminated due to suspected invasion of the system.

e) The exchange of Coded Message 2 functions may occur at any time, thus creating a keep-alive, as well as continually re-authenticating connections.

f) The encoded data in the Coded Message 2 may also be used as the encryption key by inserting the optional encryption header.

6) **Configuration Control.** The features listed may be made configurable. For example, some systems may not support encryption, while others may allow many messages to be queued before acknowledgment. Other systems may require coded messages. These should all be implemented via service levels, not by specific option enabling techniques.

7) **Precise Error and Status Reporting.** The response codes permit isolation of errors clearly and cleanly. For example, there are codes for both “Invalid Function” and “Unsupported Function”. This permits an interface to query a peer interface to determine what level of functionality is supported.

APPENDIX A – ENCRYPTION KEY BOOK FORMAT

Encryption keys will be distributed in the format defined below and referenced from within DMPP encryption headers as defined in Section 4 of this document. The key book will be ASCII text consisting of a Book ID record, Keys record and some number of Key records. The supported formats are the fixed format and the xml format as described below.

RECORD	DESCRIPTION
Book ID	Text format: • First record: book identifier • Format: "BOOK:" followed by ID • ASCII integers (0<Book ID<32K) XML Format: • Node "<BOOK>" • Attribute "ID" • Elements for the keys follow (below)
Keys	Text Format: • Second record: number of keys • Format: "KEYS:" followed by number of keys in book • ASCII integers (0<Keys<32K) XML Format: • Element "<KEYS>" • Attribute "ID" • Actual data in Hex characters
Key	Text Format: • Subsequent records: keys • Format: Key ID followed by ":" followed by key • Key ID: 5 ASCII integers with leading zeros (0<Key ID<32K) • Key: 32 ASCII-encoded hex characters for 128-bit keys; 64 characters for 256-bit keys • Key IDs must start at one and be sequential and contiguous XML Format: • Not Applicable

Example of "fixed format" key book file with 10 keys for 128 bit encryption:

```
BOOK:23
KEYS:10
00001:B8944C0CDB06DC5FD0F58C09749A44DD
00002:9FE3BF381FA0911C40464FF6422A66B5
00003:A32917E4C64EAA618BED08E6A8875640
00004:3F0657274A82FA861C7C9D03115208A8
00005:D13A752BE81C67735192E174DC4A4105
00006:C43745073D9CE581A6F1E95273F2058E
00007:86DDABF576B268D868397AE54428395E
```

```
00008:73AB9C2C3F10C671120B8837BC6EB1AB
00009:0664FCB6B456F1A51216F87F3664828F
00010:8EA7B1CB7235C08CC5FCAEE61FCB6022
```

Example of “xml format” key book file with 10 keys:

```
<BOOK ID=”23”>
  <KEYS ID=”1”>B8944C0CDB06DC5FD0F58C09749A44DD</KEY>
  <KEYS ID=”2”>9FE3BF381FA0911C40464FF6422A66B5</KEY>
  <KEYS ID=”3”>A32917E4C64EAA618BED08E6A8875640</KEY>
  <KEYS ID=”4”>3F0657274A82FA861C7C9D03115208A8</KEY>
  <KEYS ID=”5”>D13A752BE81C67735192E174DC4A4105</KEY>
  <KEYS ID=”6”>C43745073D9CE581A6F1E95273F2058E</KEY>
  <KEYS ID=”7”>86DDABF576B268D868397AE54428395E</KEY>
  <KEYS ID=”8”>73AB9C2C3F10C671120B8837BC6EB1AB</KEY>
  <KEYS ID=”9”>0664FCB6B456F1A51216F87F3664828F</KEY>
  <KEYS ID=”10”>8EA7B1CB7235C08CC5FCAEE61FCB6022</KEY>
</BOOK>
```