DOCUMENT 123-20

Range Commanders Council

Telemetry Group

IRIG 106 CHAPTER 10 PROGRAMMERS' HANDBOOK

ABERDEEN TEST CENTER
DUGWAY PROVING GROUND
ELECTRONIC PROVING GROUND
REAGAN TEST SITE
REDSTONE TEST CENTER
WHITE SANDS TEST CENTER
YUMA PROVING GROUND

NAVAL AIR WARFARE CENTER AIRCRAFT DIVISION PATUXENT RIVER
NAVAL AIR WARFARE CENTER WEAPONS DIVISION CHINA LAKE
NAVAL AIR WARFARE CENTER WEAPONS DIVISION POINT MUGU
NAVAL SURFACE WARFARE CENTER DAHLGREN DIVISION
NAVAL UNDERSEA WARFARE CENTER DIVISION KEYPORT
NAVAL UNDERSEA WARFARE CENTER DIVISION NEWPORT
PACIFIC MISSILE RANGE FACILITY

30TH SPACE WING
45TH SPACE WING
96TH TEST WING
412TH TEST WING
ARNOLD ENGINEERING DEVELOPMENT COMPLEX

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

This page intentionally left blank.

DOCUMENT 123-20

IRIG 106 PROGRAMMERS HANDBOOK

August 2020

Prepared by
TELEMETRY GROUP
RANGE COMMANDERS COUNCIL

Published by
Secretariat
Range Commanders Council
U.S. Army White Sands Missile Range,
New Mexico 88002-5110

This page intentionally left blank.

Table of Contents

Table of Figures

Table of Tables

Foreword

This handbook is intended to assist programmers to develop software for use with IRIG 106 standard instrumentation recorders. This handbook primarily addresses IRIG 106 Chapter 10 recorders, but also covers aspects of Chapter 6 and Chapter 9.

Task Lead: Mr. James P. Ferrill
Instrumentation Division
412 Test Engineering Group
307 Popson Dr.
Edwards AFB, CA 93524
E-Mail: james.ferrill.ctr@us.af.mil

Please direct any questions to:

Secretariat, Range Commanders Council
ATTN: TEWS-RCC
1510 Headquarters Avenue
White Sands Missile Range, New Mexico 88002-5110
Telephone: (575) 678-1107, DSN 258-1107
E-Mail: rcc-feedback@trmc.osd.mil

This page intentionally left blank.

Acronyms

ACTTS	Air Combat Test and Training System
Ack	Acknowledgement
ACMI	Air Combat Maneuvering Instrumentation
ACTS	Air Combat Training System
ANSI	American National Standards Institute
API	application programming interface
ASCII	American Standard Code for Information Interchange
AVC	Advanced Video Coding
CAN	controller area network
CDB	Command Descriptor Block
CSDW	channel-specific data word
CTS	Combat Training System
DCRsi	Digital Cartridge Recording System
EAG	European Air Group
ERTC	Extended Relative Time Counter
FCP	Fibre Channel Protocol
FC-PLDA	Fibre Channel Private Loop SCSI Direct Attach
GCC	Gnu Compiler Collection
GPS	Global Positioning System
IAW	in accordance with
ICD	interface control document
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
I/O	input/output
IOCTL	input/output control code
IP	Internet Protocol
IPDH	intra-packet data header
IPH	intra-packet header
IPTS	intra-packet time stamp
IRIG	Inter-Range Instrumentation Group
iSCSI	Internet Small Computer Systems Interface
IU	information unit
KITS	Kadena Interim Training System
KLV	key-length-value
LBA	logical block address
LUN	logical unit number
MAC	media access control
MBR	master boot record
MIL-STD	Military Standard
MTU	maximum transfer unit
NMEA	National Marine Electronics Association
NTP3	Network Time Protocol 3
PCM	pulse code modulation
PDU	protocol data unit

PHY	Physical Layer
PS	program stream
RCC	Range Commanders Council
RFC	Request for Comment
RIU	remote interface unit
RMM	removable memory module
RTC	relative time counter
RTCM	Radio Technical Communication for Maritime Services
SBP	Serial Bus Protocol
SCSI	Small Computer Systems Interface
SPT	SCSI Pass Through
STANAG	Standardization Agreement
TCP	Transmission Control Protocol
TMATS	Telemetry Attributes Transfer Standard
TS	transport stream
TSPI	time-space-position information
UART	Universal Asynchronous Receiver and Transmitter
UDP	User Datagram Protocol
XML	eXtensible Markup Language
XSD	XML schema definition

CHAPTER 1

Scope

1.1 General

The Telemetry Group of the Range Commanders Council (RCC) has developed the Inter-Range Instrumentation Group (IRIG) 106[1] standard for test range telemetry. The purpose of IRIG 106 is to define a common framework for test range instrumentation primarily to ensure test range interoperability. The RCC periodically revises and reissues IRIG 106. A specific version of IRIG 106 is suffixed with the last two digits of the year it was released. For example IRIG 106-07 refers to the version of IRIG 106 released in 2007.

The IRIG 106 is composed of 19 chapters, each devoted to a different element of the telemetry system or process. One of the major topics of the IRIG 106 standard is Chapter 10, the Digital Recording Standard.[2] Chapter 10 defines the interfaces and operational requirements for digital data recording devices. Chapter 10 also references elements of Chapter 6, Recorder & Reproducer Command and Control[3], and Chapter 9, Telemetry Attributes Transfer Standard[4] (TMATS).

Chapter 10 is quite comprehensive in its scope. The purposes of this handbook are to serve as an adjunct to the IRIG 106 standard to help the computer programmer write software for operating IRIG 106 Chapter 10 standard digital recorders, and to analyze data from these recorders. A prior working knowledge of Chapters 9 and 10, as well as applicable sections of Chapter 6 is essential.

1.2 Document Layout

This document addresses specific topics of Chapters 6, 9, and 10 important to the programmer. Algorithms and data structures are presented to assist the programmer in correctly interpreting IRIG 106 and implementing software for use with digital data recorders. In particular, data structures are defined in American National Standards Institute (ANSI) C for data defined in IRIG 106. Guidance is also offered on specific programming techniques.

Chapter 2 covers reading and interpreting Chapter 9 recorder configuration files.

Chapter 3 covers data retrieval over the standard recorder interfaces.

Chapter 4 covers recorder command and control.

---

[1] Range Commanders Council. *Telemetry Standards*. IRIG 106-20. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/106-20_Telemetry_Standards.pdf.

[2] Range Commanders Council. "Digital Recording Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 10. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/chapter10.pdf.

[3] Range Commanders Council. "Recorder & Reproducer Command and Control" in *Telemetry Standards*. IRIG 106-20 Chapter 6. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/chapter6.pdf.

[4] Range Commanders Council. "Telemetry Attributes Transfer Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 9. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/Chapter9.pdf.

Chapter 5 covers data file interpretation.

Chapter 6 covers IRIG 106 standard conformance issues.

Appendix B, Appendix C, and Appendix D offer example source code and a mapping between Chapter 10 files and an XML format.

1.3 Document Conventions

In the sections that follow, example computer source code and data structures are presented. All computer code is written in ANSI C. Occasionally C-like pseudo-code is used to demonstrate an algorithm more succinctly than strictly legal C. These instances will be obvious from the context.

Different programming languages have different default-sized variables. Even different compilers for a single language like C will have different variable sizes. Many variables and structures need to be represented with specific-sized variables. In this document and with the source code that accompanies it, this is accomplished by defining standard-sized variable types. The variable type naming convention used is the same convention used in later versions of the Gnu Compiler Collection (GCC) C run-time library. The variable type names used are shown in Table 1-1.

Table 1-1. Standard-Sized Variable Types

`int8_t`	integer, signed, 8 bit
`int16_t`	integer, signed, 16 bit
`int32_t`	integer, signed, 32 bit
`int64_t`	integer, signed, 64 bit
`uint8_t`	integer, unsigned, 8 bit
`uint16_t`	integer, unsigned, 16 bit
`uint32_t`	integer, unsigned, 32 bit
`uint64_t`	integer, unsigned, 64 bit

Hungarian notation is used for variable and structure naming to help keep variable type and meaning clear. The Hungarian prefixes used in the example code are shown in Table 1-2.

Table 1-2. Hungarian Notation Prefixes

`i`	Signed integer
`u`	Unsigned integer
`b`	Boolean flag
`ch`	Character
`by`	Byte
`su`	Structure variable
`Su`	Structure name
`a`	Array of...
`p`	Pointer to...

CHAPTER 2

Recorder Setup and Configuration

Chapter 9 of the IRIG 106 defines TMATS, which historically has been used as a shorthand way of documenting and describing recorded data to facilitate future data interpretation and reduction. In the context of Chapter 10, TMATS is still used to document recorded data, but is also used for recorder setup and configuration.

The TMATS format is designed to be humanly readable text, but structured to be machine-parsable. Each attribute appears in the TMATS file as a unique code name and data item pair. The code name appears first, delimited by a colon. The data item follows, delimited by a semicolon. An attribute has the form "**`CODE:DATA;`**". Lines may be terminated with a carriage return and line feed to improve readability, but are not required. Attributes are limited to 2048 bytes. The 2007 version of the IRIG 106 introduced an XML version of TMATS, although vendor support for this method has been limited to this point.

The TMATS attributes are logically arranged in a hierarchical tree structure. Figure 9-1 in the Chapter 9 standard shows this attribute tree structure. Unlike other markup languages, eXtensible Markup Language (XML) for example, the structure of the attribute tree is not inherent in the position, structure, or syntax of individual TMATS lines. Instead the hierarchical connections are deduced by matching attribute names from different tree levels. For example, TMATS "R" attributes are linked to the corresponding TMATS "G" attribute by matching the "**`R-m\ID`**" (e.g., **`R-1\ID:MyDataSource;`**) attribute with the corresponding "**`G\DSI-n`**" attribute (e.g., **`G\DSI-1:MyDataSource;`**). An example of a portion of a TMATS attribute tree is shown in Figure 2-1. Chapter 9 defines the specific linking fields for the various levels of the TMATS attribute tree.

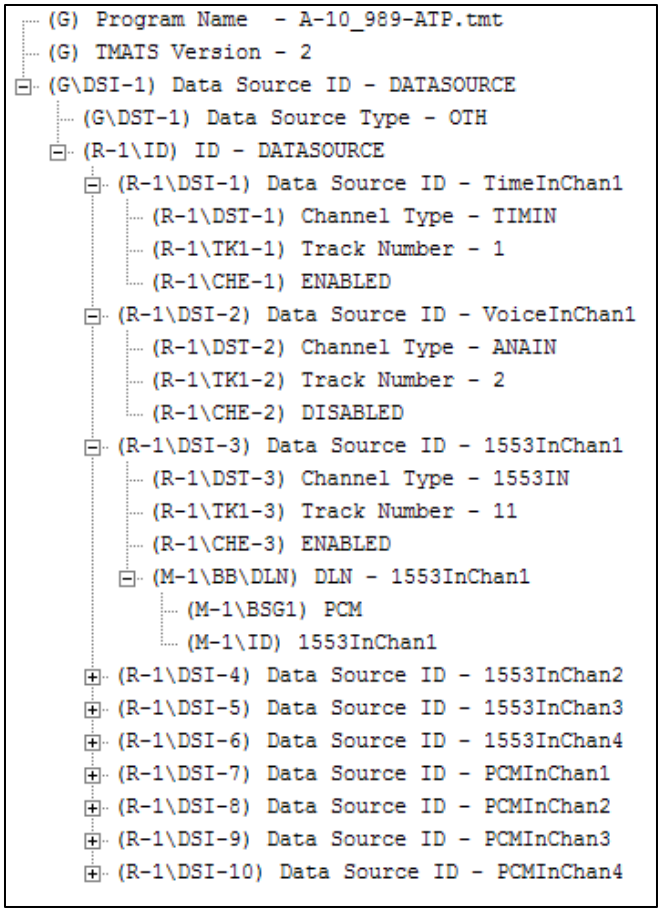

Figure 2-1. Example TMATS Attribute Tree

Attribute lines can be easily parsed using the string tokenizer function **`strtok()`** in the C run time library. An example approach outline for TMATS parsing is shown in Figure 2-2. The TMATS attribute line is stored in the null terminated character array **`szLine[]`**. The code name string is pointed to by **`szCodeName`** and the data item value is pointed to by **`szDataItem`**. Specific parsers are called for specific attribute types, indicated by the first letter of the code name. After all TMATS attributes are read, they are linked into a hierarchical tree. A more complete example of TMATS parsing is presented in Appendix D.

```
char                szLine[2048];
char              * szCodeName;
char              * szDataItem;

// Split the line into left hand and right hand sides
szCodeName = strtok(szLine, ":");
szDataItem = strtok(NULL, ";");

// Determine and decode different TMATS types
switch (szCodeName[0])
{
    case 'G' : // Decode General Information
        break;
    case 'B' : // Decode Bus Data Attributes
        break;
    case 'R' : // Decode Tape/Storage Source Attributes
        break;
    case 'T' : // Decode Transmission Attributes
        break;
    case 'M' : // Decode Multiplexing/Modulation Attributes
        break;
    case 'P' : // Decode PCM Format Attributes
        break;
    case 'D' : // Decode PCM Measurement Description
        break;
    case 'S' : // Decode Packet Format Attributes
        break;
    case 'A' : // Decode PAM Attributes
        break;
    case 'C' : // Decode Data Conversion Attributes
        break;
    case 'H' : // Decode Airborne Hardware Attributes
        break;
    case 'V' : // Decode Vendor Specific Attributes
        break;
    default :
        break;
    } // end decoding switch

    // Now link the various records together into a tree
    vConnectRtoG(...);
    vConnectMtoR(...);
    vConnectBtoM(...);
```

Figure 2-2. TMATS Attribute Parser Example Code

There are two basic types of attribute code names: single entry and multiple entry. Single-entry attributes are those for which there is only one data item and appear once in TMATS. For example:

G\PN:EW EFFECTIVENESS;

Multiple-entry attributes may appear multiple times. They are distinguished by a numeric identifier preceded by a hyphen. For example:

```
G\DSI-1:Aircraft;
G\DSI-2:Missile;
G\DSI-3:Target;
```

Some attributes can appear multiple times at multiple levels. For example, the Message Data Sub-Channel Name attribute "**R-x\MCNM-n-m**" may appear associated with multiple recorder groups ("**x**"), multiple message data channels ("**n**"), and with multiple subchannels ("**m**").

Chapter 9 identifies quite a few required TMATS attributes. These attributes are necessary to ensure correct recorder setup and subsequent data interoperability. For correct TMATS parsing and interpretation, an important required attribute is "**G\106**". This attribute identifies the version of IRIG 106 to use when interpreting the TMATS information. This attribute only identifies the TMATS version. The overall recorder version is specified in the TMATS setup record in the recorded data file described in Subsection 5.5.2.

Chapter 9 states that attributes may appear in any order. Chapter 10, however, requires some specific TMATS comment attributes follow other specific modified TMATS attributes for modified data files. This complicates TMATS machine parsing considerably. When reading and decoding TMATS, a parser must maintain the most recent recorder data channel state so that when a comment attribute is encountered, it can be associated with the correct recorder channel. When writing TMATS, the appropriate comments must follow the appropriate attribute records. See Chapter 10 for specific TMATS attribute position and order requirements.

Chapter 9 of IRIG 106-07 introduced support for TMATS to be represented in XML format. Although XML TMATS was defined in 106-07 Chapter 9, there was no way to distinguish traditional American Standard Code for Information Interchange (ASCII) TMATS from XML TMATS in an IRIG 106-07 Setup Record packet. Chapter 10 of IRIG 106-09 added a field to the Setup Record packet to indicate format of the TMATS record. The TMATS XML format is implemented as a standard XML schema consisting of a collection of XML schema definition (XSD) files. The TMATS XSDs can be viewed at the official RCC site. Although XML TMATS is maintained in parallel with traditional ASCII TMATS, XML TMATS is not widely supported in the test community.

The TMATS standard has evolved considerably from year to year. Appendix A is a chart of legal TMATS attributes by IRIG 106 release year.

CHAPTER 3

Data Retrieval

3.1 Small Computer Systems Interface (SCSI) Protocol

Recorded data from a Chapter 10 recorder is retrieved by transferring it to a host computer over one of several interfaces provided by the recorder. Chapter 10 requires that each removable memory module (RMM) provide an IEEE 1394b data download port. Chapter 10 also requires that each recorder provide either a Fibre Channel or IEEE 1394b data download port, and optionally an Ethernet download port.

The protocol for data download over the various interface types is the SCSI block transfer protocol. Overall SCSI operation is defined in the SCSI Architecture Model 2 (SAM-2)[5] document. Common commands are defined in the SCSI Primary Commands (SPC-2)[6] document. Commands for block devices are defined in the SCSI Block Commands 2 (SBC-2)[7] document.

The SCSI architecture is a client-server model. The client is the user application (the "initiator"), requesting services (status, data, etc.) from the SCSI server device (the "target"). An application client is independent of the underlying interconnect and SCSI transport protocol. Each SCSI target is identified by a target device name. Targets are addressed using the target device name. Different transports support different methods for uniquely naming SCSI targets. Each target device also supports one or more logical unit numbers (LUNs), which are used to support different services from a single SCSI target device. For example, an RMM provides disk file access using LUN 0 and real-time clock access using LUN 1.

The SCSI architecture model defines a Command Descriptor Block (CDB) structure along with associated user input/output (I/O) buffers. In order to issue a command to a SCSI device, a CDB must be used. The SCSI protocol defines a large number of commands to support a wide range of devices. The Chapter 10 standard only requires a small subset of the complete SCSI command set to be implemented to support the RMM and remote data access block transfer device. Acceptable lengths for CDBs can be 6, 10, 12, or 16 bytes. Chapter 10 currently only requires 6- and 10-byte CDBs. Note that multi-byte CDB values such as logical block address (LBA) are big-endian in the CDB and require writing to the CDB a byte at a time from a little-endian processor to write the multi-byte values in proper order.

The SCSI INQUIRY command is used to query the SCSI device about its capabilities. The structure for the INQUIRY CDB is shown in Figure 3-1. The structure for the Control field is shown in Figure 3-2. The data download interface is required to support the standard

[5] ISO/IEC. *Information Technology – Small Computer System Interface (SCSI) Part 412: Architecture Model-2 (SAM-2).* ISO/IEC 14776-412. October 2006. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/39517.html.

[6] ISO/IEC. *Information Technology – Small Computer System Interface (SCSI) Part 452: SCSI Primary Commands-2 (SPC-2).* ISO/IEC 14776-452:2005. August 2005. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/38775.html.

[7] ISO/IEC. *Information Technology – Small Computer System Interface (SCSI) Part 322: SCSI Block Commands-2 (SBC-2).* ISO/IEC 14776-322:2007. February 2007. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/42101.html.

INQUIRY response shown in Figure 3-3. Also required are the Supported Vital Product page, Unit Serial Number page, and Device Identification page.

```
struct SuCdbInquiry
    {
    uint8_t       uOpCode;              // Operation code = 0x12
    uint8_t       bEVPD         :  1;   // Enable vital product data
    uint8_t       bCmdDt        :  1;   // Command support data
    uint8_t       Reserved1     :  6;   //
    uint8_t       uPageOpCode;          // Page or operation code
    uint8_t       Reserved2;            //
    uint8_t       uAllocLength;         // Allocation length
    struct SuCdbControl  suControl;
    };
```

Figure 3-1. SCSI INQUIRY CDB Structure

```
struct SuCdbControl
    {
    uint8_t       bLink         : 1;
    uint8_t       Obsolete      : 1;
    uint8_t       bNACA         : 1;
    uint8_t       Reserved      : 3;
    uint8_t       uVendor       : 2;
    };
```

Figure 3-2. SCSI CDB Control Field Structure

```
struct SuCdbInquiryStdData
    {
    uint8_t       uPeriphType  :  5;    // Peripheral device type
    uint8_t       uPeriphQual  :  3;    // Peripheral qualifier
    uint8_t       uReserved1   :  7;
    uint8_t       bRMB         :  1;    // Removable medium
    uint8_t       uVersion;             // Version
    uint8_t       uFormat      :  4;    // Response data format
    uint8_t       bHiSup       :  1;    // Hierarchical support
    uint8_t       bNormACA     :  1;    // Support normal ACA bit
    uint8_t       uReserved2   :  1;
    uint8_t       bAERC        :  1;    // Asynch event reporting cap
    uint8_t       uAddLength;           // Length of add parameters
    uint8_t       uReserved3   :  7;
    uint8_t       bSCCS        :  1;    // Embedded storage supported
    uint8_t       bAddr16      :  1;    // Not used
    uint8_t       uReserved4   :  2;
    uint8_t       bMChngr      :  1;    // Medium changer
    uint8_t       bMultiP      :  1;    // Multi-port device
    uint8_t       bVS1         :  1;    // Vendor specific
    uint8_t       bEncServ     :  1;    // Enclosure service
    uint8_t       bBQue        :  1;    // Basic queing
    uint8_t       bVS2         :  1;    // Vendor specific
    uint8_t       bCmdQue      :  1;    // Command queuing supported
    uint8_t       uReserved5   :  1;
    uint8_t       bLinked      :  1;    // Linked commands supported
    uint8_t       bSync        :  1;    // Not used
    uint8_t       bWBus16      :  1;    // Not used
    uint8_t       uReserved6   :  1;
    uint8_t       bRelAddr     :  1;    // Relative addressing supported
    uint8_t       uVendorID[8];         //
    uint8_t       uProductID[16];       //
    uint8_t       uProductRev[4];       //
    };
```

Figure 3-3. SCSI INQUIRY Data Structure

The SCSI READ CAPACITY command is used to query the disk device about its size. The structure for the READ CAPACITY CDB is shown in Figure 3-4. This command returns the number of available logical blocks and the logical block size in bytes, shown in Figure 3-5. Note that returned values are big-endian and must be byte swapped before they can be used on a little-endian processor.

```
struct SuCdbReadCapacity10
    {
    uint8_t       uOpCode;              // Operation code = 0x25
    uint8_t       uReserved1;           //
    uint8_t       uLBA_3_MSB;           // Logical block address, MSB
    uint8_t       uLBA_2;               // Logical block address
    uint8_t       uLBA_1;               // Logical block address
    uint8_t       uLBA_0_LSB;           // Logical block address, LSB
    uint8_t       uReserved2;           //
    uint8_t       uReserved3;           //
    uint8_t       bPMI          :  1;   // Partial medium indicator
    uint8_t       uReserved3    :  7;   //
    struct SuCdbControl  suControl;
    };
```

Figure 3-4. SCSI READ CAPACITY CDB Structure

```
struct SuCdbReadCapacityData
    {
    uint64_t      uBlocks;              // Logical blocks (big endian!)
    uint64_t      uBlockSize;           // Block size (big endian!)
    };
```

Figure 3-5. SCSI READ CAPACITY Data Structure

The SCSI READ command is used to read logical blocks of data from the disk device. The SCSI protocol provides five different READ commands with various capabilities and sizes of the CDB. The Chapter 10 standard only requires the 10-byte variant of the READ command. The structure for the READ CDB is shown in Figure 3-6. This command returns the data from the requested logical blocks.

```
struct SuCdbRead10
    {
    uint8_t       uOpCode;              // Operation code = 0x28
    uint8_t       bReserved1    :  1;   //
    uint8_t       bFUA_NV       :  1;   // Force unit access non-volatile
    uint8_t       Reserved2     :  1;   //
    uint8_t       bFUA          :  1;   // Force unit access
    uint8_t       bDPO          :  1;   // Disable page out
    uint8_t       bRdProtect    :  3;   // Read protect
    uint8_t       uLBA_3_MSB;           // Logical block address, MSB
    uint8_t       uLBA_2;               // Logical block address
    uint8_t       uLBA_1;               // Logical block address
    uint8_t       uLBA_0_LSB;           // Logical block address, LSB
    uint8_t       uGroupNum     :  5;   // Group number
    uint8_t       Reserved3     :  3;   //
    uint8_t       uTransLength_1_MSB;   // Transfer length, MSB
    uint8_t       uTransLength_0_LSB;   // Transfer length, LSB
    struct SuCdbControl  suControl;
    };
```

Figure 3-6. SCSI READ(10) CDB Structure

3.1.1 Institute of Electrical and Electronics Engineers (IEEE) 1394

The IEEE 1394[8] standard defines the Serial Bus Protocol (SBP-2) for transporting CDBs and data over a 1394 bus to a SCSI target. The basic unit of information in SBP-2 is the Operation Request Block, which encapsulates the information in SCSI CDBs and I/O buffers, as well as additional information needed by the 1394 bus. 1394 operates by exposing a shared memory address space, negotiated during device initialization. The CDBs carried within SBP packets are written to the SCSI target device shared memory, and associated data is written to and read from shared memory.

When interfacing to an RMM, LUN 0 is used for disk access. The LUN 1 is used for interfacing to the RMM real-time clock.

When interfacing to a recorder, LUN 0 or LUN 32 is used for disk access.

3.1.2 Fibre Channel

Fibre Channel defines the Fibre Channel Protocol (FCP) for transporting CDBs and data over a Fibre Channel bus to a SCSI target. The basic unit of information in FCP is the information unit (IU). Communication with SCSI targets uses several types of IUs defined by FCPs. Fibre Channel also defines the Fibre Channel Private Loop SCSI Direct Attach (FC-PLDA) protocol, which further defines the implementation of SCSI over Fibre Channel. Chapter 10 requires conformance to FC-PLDA.

3.1.3 Internet Small Computer Systems Interface

The Internet Engineering Task Force (IETF) has published Request for Comment (RFC) 3270[9] defining the Internet Small Computer Systems Interface (iSCSI) protocol for transporting CDBs over Transmission Control Protocol (TCP)/Internet Protocol (IP)-based networks to a SCSI target. Chapter 10 identifies iSCSI as the standard protocol for recorder data access over Ethernet. The basic unit of information in iSCSI is the protocol data unit (PDU), which encapsulates the information in SCSI CDBs and I/O buffers as well as additional information needed by the underlying IP network. The PDUs are transported over a TCP connection, usually using well-known port number 860 or 3260. The actual port number used is not specified in Chapter 10.

For an iSCSI initiator to establish an iSCSI session with an iSCSI target, the initiator needs the IP address, TCP port number, and iSCSI target name information. There are several methods that may be used to find targets. The iSCSI protocol supports the following discovery mechanisms.

- Static Configuration: This mechanism assumes that the IP address, TCP port, and the iSCSI target name information are already available to the initiator. The initiators need to perform no discovery in this approach. The initiator uses the IP address and the TCP port

[8] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High-Performance Serial Bus*. IEEE 1394-2008. New York: Institute of Electrical and Electronics Engineers, 2008.
[9] Internet Engineering Task Force. "Multi-Protocol Label Switching (MPLS) Support of Differentiated Services." RFC 3270. May 2002. Updated by RFC 5462. Retrieved 12 August 2019. Available at http://datatracker.ietf.org/doc/rfc3270/.

information to establish a TCP connection, and it uses the iSCSI target name information to establish an iSCSI session. This discovery option is convenient for small iSCSI setups.

- SendTargets: This mechanism assumes that the target's IP address and TCP port information are already available to the initiator. The initiator then uses this information to establish a discovery session to the network entity. The initiator then subsequently issues the SendTargets text command to query information about the iSCSI targets available at the particular network entity (IP address).
- Zero-Configuration: This mechanism assumes that the initiator does not have any information about the target. In this option, the initiator can either multicast discovery messages directly to the targets or it can send discovery messages to storage name servers. Currently, there are many general-purpose discovery frameworks available. Service Location Protocol (RFC 2608[10]) and Internet Storage Name Service (RFC 4171[11]) are two popular discovery protocols.

Target discovery is not specified in Chapter 10.

When interfacing to a recorder, LUN 0 or LUN 32 is used for disk access while LUN 1 or LUN 33 is used for command and control.

3.2 Software Interface

All recorder data download interfaces appear as SCSI block I/O devices and respond to the subset of SCSI commands set forth in Chapter 10, but different operating systems provide vastly different types of application programming interfaces (APIs) for communicating with recorders and RMMs over the various data download interfaces specified in Chapter 10.

The Microsoft Windows device driver environment helps remove a lot of complexity from communicating over the various data interfaces. Windows Plug and Play drivers are able to discover various types of SCSI devices connected to them, initialize and configure them, and then offer a single programming interface for user application programs.

The interface used by Windows applications to send SCSI commands to a SCSI device is called SCSI Pass Through (SPT). Windows applications can use SPT to communicate directly with SCSI devices using the Win32 API **`DeviceIoControl()`** call and the appropriate I/O control code (IOCTL).

Before any IOCTLs can be sent to a SCSI device, a handle for the device must be obtained. The Win32 API **`CreateFile()`** is used to obtain this handle and to define the sharing mode and the access mode. The access mode must be specified as (GENERIC_READ | GENERIC_WRITE). The key to obtaining a valid handle is to supply the proper filename for the device that is to be opened.

For Chapter 10 SCSI devices, the SCSI class driver defines an appropriate name. If the device is unclaimed by a SCSI class driver (the usual case), then a handle to the SCSI port driver is required. The filename in this case is "**`\\.\ScsiN:`**", where **`N`** = 0, 1, 2, etc. The number **`N`** corresponds to the SCSI host adapter card number that controls the desired SCSI device. When

[10] Internet Engineering Task Force. "Service Location Protocol, Version 2." RFC 2608. June 1999. Updated by RFC 3224. Retrieved 12 August 2019. Available at http://datatracker.ietf.org/doc/rfc2608/.
[11] Internet Engineering Task Force. "Internet Storage Name Service (iSNS)." RFC 4171. September 2005. May be superseded by update. Retrieved 12 August 2019. Available at https://datatracker.ietf.org/doc/rfc4171/.

the SCSI port name is used, the Win32 application must set the proper PathId, TargetId, and LUN in the SPT structure.

Once a valid handle to a SCSI device is obtained, then appropriate I/O buffers for the requested IOCTL must be allocated and, in some cases, filled in correctly.

There are several IOCTLs that the SCSI port driver supports, including the following.

- IOCTL_SCSI_GET_INQUIRY_DATA
- IOCTL_SCSI_GET_CAPABILITIES
- IOCTL_SCSI_PASS_THROUGH
- IOCTL_SCSI_PASS_THROUGH_DIRECT

IOCTL_SCSI_GET_INQUIRY_DATA returns a SCSI_ADAPTER_BUS_INFO structure for all devices that are on the SCSI bus. The structure member, BusData, is a structure of type SCSI_BUS_DATA. It contains an offset to the SCSI Inquiry data, which is also stored as a structure, SCSI_INQUIRY_DATA.

Within the SCSI_INQUIRY_DATA is a structure member named **DeviceClaimed**, which indicates whether or not a class driver has claimed this particular SCSI device. If a device is claimed, all SPT requests must be sent first through the class driver, which will typically pass the request unmodified to the SCSI port driver. If the device is unclaimed, the SPT requests are sent directly to the SCSI port driver.

For IOCTL_SCSI_GET_INQUIRY_DATA, data is only read from the device and not sent. Set lpInBuffer to NULL and nInBufferSize to zero. The output buffer might be quite large, as each SCSI device on the bus will provide data that will fill three structures for each device: SCSI_ADAPTER_BUS_INFO, SCSI_BUS_DATA, and SCSI_INQUIRY_DATA. Allocate a buffer that will hold the information for all the devices on that particular SCSI adapter. Set lpOutBuffer to point to this allocated buffer and nOutBufferSize to the size of the allocated buffer.

IOCTL_SCSI_GET_CAPABILITIES returns an IO_SCSI_CAPABILITIES structure. This structure contains valuable information about the capabilities of the SCSI adapter. Two items of note are the MaximumTransferLength, which is a byte value indicating the largest data block that can be transferred in a single SCSI Request Block, and the MaximumPhysicalPages, which is the maximum number of physical pages that a data buffer can span.

The two IOCTLs IOCTL_SCSI_PASS_THROUGH and IOCTL_SCSI_PASS_THROUGH_DIRECT allow SCSI CDBs to be sent from a Win32 application to a SCSI device. Depending on which IOCTL is sent, a corresponding pass through structure is filled out by the Win32 application. IOCTL_SCSI_PASS_THROUGH uses the structure SCSI_PASS_THROUGH. IOCTL_SCSI_PASS_THROUGH_DIRECT uses the structure SCSI_PASS_THROUGH_DIRECT. The SCSI_PASS_THROUGH structure is shown in Figure 3-7.

```
struct SCSI_PASS_THROUGH
    {
    USHORT Length;
    UCHAR ScsiStatus;
    UCHAR PathId;
    UCHAR TargetId;
    UCHAR Lun;
    UCHAR CdbLength;
    UCHAR SenseInfoLength;
    UCHAR DataIn;
    ULONG DataTransferLength;
    ULONG TimeOutValue;
    ULONG DataBufferOffset;
    ULONG SenseInfoOffset;
    UCHAR Cdb[16];
    }
```

Figure 3-7. SCSI_PASS_THROUGH Structure

The structures SCSI_PASS_THROUGH and SCSI_PASS_THROUGH_DIRECT are virtually identical. The only difference is that the data buffer for the SCSI_PASS_THROUGH structure must be contiguous with the structure. This structure member is called DataBufferOffset and is of type ULONG. The data buffer for the SCSI_PASS_THROUGH_DIRECT structure does not have to be contiguous with the structure. This structure member is called **DataBuffer** and is of type **PVOID**.

For the two SPT IOCTLs, IOCTL_SCSI_PASS_THROUGH and IOCTL_SCSI_PASS_THROUGH_DIRECT, both **lpInBuffer** and **lpOutBuffer** can vary in size depending on the Request Sense buffer size and the data buffer size. In all cases, **nInBufferSize** and **nOutBufferSize** must be at least the size of the SCSI_PASS_THROUGH (or SCSI_PASS_THROUGH_DIRECT) structure. Once the appropriate I/O buffers have been allocated, then the appropriate structure must be initialized.

The **Length** is the size of the SCSI_PASS_THROUGH structure. The **ScsiStatus** should be initialized to 0. The SCSI status of the requested SCSI operation is returned in this structure member. The **PathId** is the bus number for the SCSI host adapter that controls the SCSI device in question. Typically, this value will be 0, but there are SCSI host adapters that have more than one SCSI bus on the adapter. The **TargetId** and **Lun** are the SCSI ID number and LUN for the device.

The **CdbLength** is the length of the CDB. Typical values are 6, 10, and 12 up to the maximum of 16. The **SenseInfoLength** is the length of the **SenseInfo** buffer. **DataIn** has three possible values; SCSI_IOCTL_DATA_OUT, SCSI_IOCTL_DATA_IN, and SCSI_IOCTL_DATA_UNSPECIFIED. The **DataTransferLength** is the byte size of the data buffer. The **TimeOutValue** is the length of time, in seconds, until a time-out error should occur. This can range from 0 to a maximum of 30 minutes (1800 seconds).

The **DataBufferOffset** is the offset of the data buffer from the beginning of the pass through structure. For the SCSI_PASS_THROUGH_DIRECT structure, this value is not an offset, but rather is a pointer to a data buffer. The **SenseInfoOffset** is similarly an offset to

the **SenseInfo** buffer from the beginning of the pass through structure. Finally, the 16 remaining bytes are for the CDB data. The format of this data must conform to the SCSI-2 standard (ISO/IEC 2006).

3.3 STANAG 4575 Directory

Chapter 10 recorders make their data available for downloading over one of their supported download ports. To the application program, the data download port appears as a block I/O disk device at SCSI LUN 0. The directory structure on the disk is a modified version of the STANAG 4575 file structure definition. The STANAG file system has been developed to enable the downloading of very large sequential files into support workstations. It supports only logically contiguous files in a single directory. The data can be physically organized any way appropriate to the media, including multiple directories, as long as the interface to the NADSI sees a single directory of files in contiguous logical addresses.

Disk blocks are accessed by LBA. It is common in many operating systems and disk structures for block 0 to be a master boot record (MBR), which typically contains operating system boot code and/or information for dividing a disk device into multiple partitions. Chapter 10 does not support MBRs or partitions. Block 0 is considered reserved, and its contents are undefined and unused.

The beginning of the STANAG directory is always read from LBA 1. The complete disk directory may span multiple disk blocks. Directory blocks are organized as a double linked list of blocks. Other than the first directory block, subsequent directory blocks can be any arbitrary block number.

A STANAG 4575 directory block has the structure shown in Figure 3-8. Keep in mind that STANAG 4575 data is big-endian and so multi-byte values will need to be byte-swapped before they can be used on a little-endian processor such as an Intel x86 found in desktop computers. The various fields in the directory block are covered in detail in the Chapter 10 standard. The **asuFileEntry[]** array holds information about individual files. Its structure is shown in Figure 3-9. The size of the **asuFileEntry[]** array will vary depending on the disk block size. For a size of 512 bytes per disk block, the **asuFileEntry[]** array will have four elements.

```
struct SuStanag4575DirBlock
    {
    uint8_t       achMagicNumber[8]     // "FORTYtwo"
    uint8_t       uRevNumber;           // IRIG 106 Revision number
    uint8_t       uShutdown;            // Dirty shutdown
    uint16_t      uNumEntries;          // Number of file entries
    uint32_t      uBlockSize;           // Bytes per block
    uint8_t       achVolName[32];       // Volume Name
    uint64_t      uFwdLink;             // Forward link block
    uint64_t      uRevLink;             // Reverse link block
    struct SuStanag4575FileBlock  asuFileEntry[4];
    };
```

Figure 3-8. STANAG 4575 Directory Block Structure

```
struct SuStanag4575FileBlock
    {
    uint8_t       achFileName[56];      // File name
    uint64_t      uFileStart;           // File start block addr
    uint64_t      uFileBlkCnt;          // File block count
    uint64_t      uFileSize;            // File size in bytes
    uint8_t       uCreateDate[8];       // File create date
    uint8_t       uCreateTime[8];       // File create time
    uint8_t       uTimeType;            // Date and time type
    uint8_t       achReserved[7];       //
    uint8_t       uCloseTime[8];        // File close time
    };
```

Figure 3-9. STANAG 4575 File Entry Structure

A complete disk file directory is read starting at LBA 1. The first directory block is read and all file entries in that block are read and decoded. Then the next directory block, LBA equal to the value in **`uFwdLink`**, is read and decoded. Directory reading is finished when the **`uFwdLink`** is equal to the current LBA. This algorithm is shown in Figure 3-10.

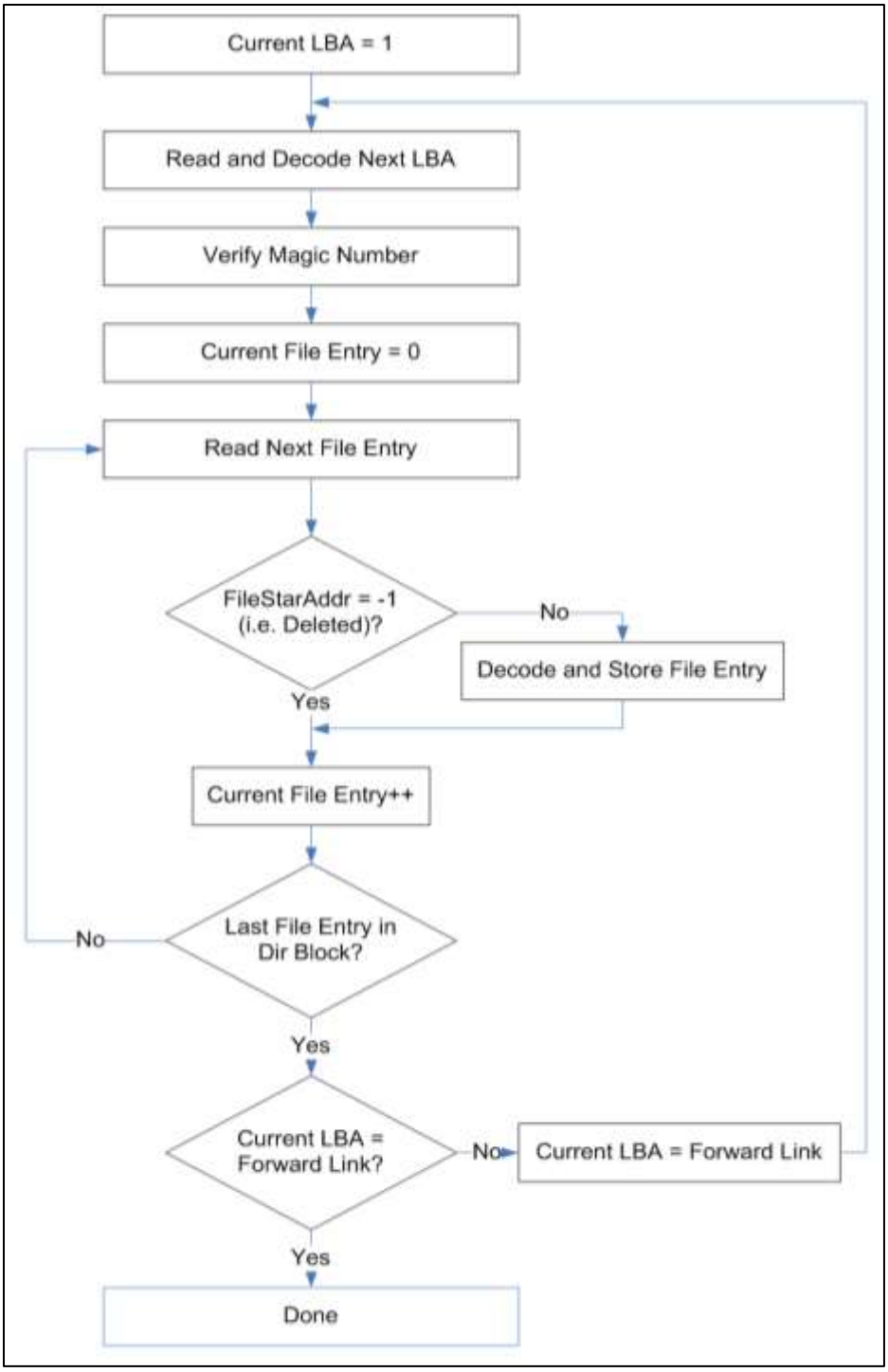

Figure 3-10. STANAG 4575 Directory Reading and Decoding Algorithm

This page intentionally left blank.

CHAPTER 4

Recorder Control

Recorders are controlled over a full duplex communications channel. Typically this channel is an RS-232 or RS-422 serial communications port. Chapter 10 also allows control over a recorder network channel.

The recorder command and control mnemonics language is defined in Chapter 6, with further requirements in Chapter 10. Interaction with a recorder is in a command/control fashion. That is, an external controller issues commands to a recorder over the command and control channel, and the recorder issues a single status response message.

Some commands, such as **`.BIT`**, can take a significant amount of time to complete. The proper method to determine when these commands complete is to issue multiple **`.STATUS`** commands, checking the status complete field until it indicates the command has completed processing.

4.1 Serial Control

Commands written to a recorder should be terminated with carriage return and line feed characters. Responses from the recorder also terminate with carriage return and line feed. In general, it's considered good defensive programming practice to recognize either character alone or both together as a valid line terminator.

Neither Chapter 6 nor Chapter 10 address serial flow control. Most commands generate very little text and buffer overflow shouldn't be a problem. The `.TMATS` command, however, can result in a considerable amount of text to be transferred. Hardware flow control requires additional signal lines that may not be available on a recorder command and control interface. It would be prudent to be prepared to support `XON` and `XOFF` flow control in software.

4.2 Network Control

Chapter 10 provides a mechanism for remote command and control over a network port using the SCSI protocol. Chapter 10 defines separate logical units for command and control. The SCSI SEND command (command code = 0x0A) along with a buffer containing a valid Chapter 6 command is used to send commands remotely. The SCSI RECEIVE command (command code = 0x08) is used to receive the command response. These SCSI commands are described in the SCSI Primary Commands document.

The IRIG 106-09 standard introduced support for the Telnet protocol for network-based recorder control. Telnet was added as a new interface type for ground-based recorders. Telnet support is mandatory for ground-based recorders but optional for airborne recorders. Note that

for network-based control and status a recorder is required to implement the Telnet standard defined in RFC 854[12], RFC 855[13], and RFC 1184[14] on TCP port 10610.

Although the full Telnet protocol is required for recorders, it is common for network client software to connect to a recorder Telnet port via a simple TCP port connection without the Telnet protocol negotiation.

[12] Internet Engineering Task Force. "Telnet Protocol Specification." RFC 854. May 1983. Updated by RFC 5198. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc854/.
[13] Internet Engineering Task Force. "Telnet Option Specifications." RFC 855. May 1983. May be superseded by update. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc855/.
[14] Internet Engineering Task Force. "Telnet Linemode Option." RFC 1184. March 2013. May be superseded by update. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc1184/.

CHAPTER 5

Data File Interpretation

5.1 Overall Data File Organization

Chapter 10 data files are organized as a sequential series of data packets. Data packets can only contain one type of data (i.e., 1553, pulse code modulation [PCM], etc.). Data packets frequently contain multiple individual data messages.

The Chapter 10 standard requires that the first data packet (or packets) in a data file be an IRIG 106 Chapter 9 format TMATS packet, which is used to configure the recorder and to describe the data begin recorded. The TMATS information is stored in a Computer-Generated Data, Format 1 (Data Type = 0x01) data packet and is discussed in Subsection 5.5.2. An important field in the packet header for the TMATS packet is the "IRIG 106 Chapter 10 Version" field. The value of this field determines the specific version of the Chapter 10 standard to use when interpreting the rest of the data file; however, this field is only defined for the 106-07 and later versions of the IRIG 106 standard. Very long TMATS setup record packets may span multiple Chapter 10 packets. This case is uncommon but is legal. Software for reading a Chapter 10 data file should be prepared to handle the case where multiple TMATS setup record packets are found at the beginning of a recording.

It is required that a time data packet be the first dynamic data packet. Dynamic data packets are not defined in the Chapter 10 standard but it is generally understood to mean any data packet other than Computer-Generated Data, Format 1 (setup record). The purpose of this requirement is to allow the association of clock time with the relative time counter (RTC) before encountering the first data packet in the data file. Programmers are cautioned, though, to verify a valid time packet has been received before attempting to interpret the RTC.

A root index packet will be the last data packet in the data file if file indexing is used. The presence of data file indexing is indicated in the TMATS setup record.

The size of all data packets is an integer multiple of 4 bytes (32 bits). Padding bytes are added, if necessary, to the end of a data packet, just before the checksum, to provide this 4-byte alignment. Regardless, when defining data structures representing Chapter 10 data packets and messages, these structures should be forced to be byte-aligned by using the appropriate compiler directive.

Some data packet elements are two or more bytes in length. For example, the first data element of data packet is a two-byte sync pattern. Multiple-byte data elements are stored in little-endian format. That is, the least significant portion of the data is stored at the lowest byte offset.

Data packets are written to disk roughly in the time order that they are received, but data packets and data messages can occur in the data file out of time order. This can occur because data recorders receive data simultaneously on multiple channels, each channel buffering data for a period of time and then writing it to disk. Because of this, individual data messages will in general be somewhat out of time order because of grouping by channel. Consider the case of two 1553 channels recording the same bus at the same time in an identical fashion. Each channel receives, buffers, and writes data to disk. The first channel will write its buffered data to disk followed by the second channel. The data from the second channel will be from the same time

period as the data from the first channel and will have identical time stamps but will be recorded after the first channel in the data file.

Starting with the IRIG 106-05 standard, recorders are only allowed to buffer data for a maximum of 100 milliseconds and data packets must be written to disk within one second. This ensures that data packets can only be out of time order by a maximum of one second. Be warned, though, that the maximum amount of time data packets can be out of order for data files produced before IRIG 106-05 is unbounded and it is not unusual to encounter data files with data packets five or more seconds out of time order.

Example source code that demonstrates basic parsing of Chapter 10 data files can be found in Appendix B. An example program that demonstrates reading and interpreting a Chapter 10 file can be found in Appendix C.

5.2 Overall Data Packet Organization

Overall data packet organization is shown in Figure 5-1. Data packets contain a standard header, a data payload containing one or multiple data messages, and a standard trailer. The standard header is composed of a required header, optionally followed by a secondary header. The data payload generally consists of a channel-specific data word (CSDW) record followed by one or more data messages.

PACKET SYNC PATTERN	Packet Header
CHANNEL ID	
PACKET LENGTH	
DATA LENGTH	
DATA VERSION	
SEQUENCE NUMBER	
PACKET FLAGS	
DATA TYPE	
RELATIVE TIME COUNTER	
HEADER CHECKSUM	
TIME	Packet Secondary Header (Optional)
RESERVED	
SECONDARY HEADER CHECKSUM	
CHANNEL SPECIFIC DATA WORD	Packet Body
INTRA-PACKET TIME STAMP 1	
INTRA-PACKET DATA HEADER 1	
DATA 1	
:	
INTRA-PACKET TIME STAMP n	
INTRA-PACKET DATA HEADER n	
DATA n	
DATA CHECKSUM	Packet Trailer

Figure 5-1. Data Packet Organization

Data packets must contain data. They are not allowed to only contain filler, although filler can be inserted into a data packet in the packet trailer before the checksum. This filler is

used to ensure data packet alignment on a four-byte boundary. Filler is also sometimes used to keep packets from a particular channel the same length. The standard does not expressly prohibit filler after the packet trailer but before the next data packet header, but inserting filler after the last trailer is considered bad practice. Still, when reading data packets, set read buffer sizes based on the value of the overall packet length found in the header. Do not make assumptions about packet length based on the data length or from information in the data payload.

When reading linearly through a Chapter 10 data file, maintaining synchronization with data packet boundaries is accomplished by using the packet length field in the header to read the appropriate amount of data or to reposition the read pointer to the beginning of the next header. In this case it is sufficient to check the value of the `Sync` field at the beginning of the header to ensure the read pointer was positioned to the beginning of a data packet.

If there is an error in the data file or if the read pointer is repositioned to some place other than the beginning of a data packet (for example to jump to the middle of a recorded data file), then the beginning of a valid data packet must be found. Unfortunately the Chapter 10 standard does not provide a way to definitively determine the beginning of a data packet in these instances. Instead some heuristics must be applied.

- Read the data file until the packet sync pattern (0xEB25) is found. (Normally the first character of the packet sync pattern is found at a file offset that is an integer multiple of four. If the data file is corrupted then the sync pattern may not fall on the normal four-byte boundary. For the best results scan the file a byte at a time, ignoring the normal four-byte alignment.)
- When the Sync pattern is found then calculate and test the header checksum.
- If a secondary header exists, calculate and test the secondary header checksum.
- Calculate and test the data checksum.

When the packet sync pattern is found and all available checksums have been verified, then there is a high probability that the beginning of the next valid data packet has been found.

5.3 Required header

The packet header contains information about the data payload such as time, packet length, data type, data version, and other information. The layout of a Chapter 10 packet header is shown in Figure 5-2.

```
struct SuI106Ch10Header
    {
    uint16_t      uSync;               // Packet Sync Pattern
    uint16_t      uChID;               // Channel ID
    uint32_t      ulPacketLen;         // Total packet length
    uint32_t      ulDataLen;           // Data length
    uint8_t       ubyHdrVer;           // Header Version
    uint8_t       ubySeqNum;           // Sequence Number
    uint8_t       ubyPacketFlags;      // PacketFlags
    uint8_t       ubyDataType;         // Data type
    uint8_t       aubyRefTime[6];      // Reference time
    uint16_t      uChecksum;           // Header Checksum
    };
```

Figure 5-2. Packet Header Structure

The Channel ID field uniquely identifies the source of the data. The value of the Channel ID field corresponds to the Track Number value of the TMATS "R" record. This field was extended to five characters in the 2009 release of IRIG 106.

Typically only one packet data type is associated with a particular Channel ID, but this is not a requirement of the Chapter 10 standard. An exception to this is Channel ID = 0, the Channel ID used for internal, computer-generated format data packets. Prior to IRIG 106-13 it was typical for Channel ID 0 to contain Computer-Generated Data, Format 0 setup records (0x01), Computer-Generated Data, Format 1 recording events records (0x02), and Computer-Generated Data, Format 3 recording index records (0x03).

With the release of IRIG 106-13 Channel 0 has been restricted to Computer-Generated Data, Format 0 (TMATS setup) records only. With the release of IRIG 106-17 Channel 0 can also contain Computer-Generated Data, Format 4 (TMATS Streaming Configuration) records. Furthermore, any other channel containing a particular format of computer-generated data is only allowed to have packets of that format.

The data payload format is interpreted based on the value of the Data Type field and the Data Version field (sometimes incorrectly called Header Version) in the packet header. Each packet data payload can only contain one type (e.g., 1553, PCM, etc.) of data. A Chapter 10 standard release will only contain data format and layout information for the latest Data Version. The specific Data Version defined in a particular Chapter 10 release can be found in the "Data Type Names and Descriptions" table. Be warned that future Chapter 10 releases may update or change data format or layout, indicated by a different Data Version value in the header, but the Chapter 10 release will not have information about the previous Data Versions. That information can only be found in the previous Chapter 10 releases.

When processing a data file, it is common to only read the data packet header, determine if the data portion is to be read (based on packet type or other information gleaned from the header), and if not to be read skip ahead to the next header. Skipping the data portion and jumping ahead to the next header is accomplished by using the packet length in the packet header. Below is the algorithm for determining how many bytes to jump ahead in the file byte stream to reposition the read pointer to the beginning of the next header.

- Read the current primary header.
- Determine relative file offset to the next header.

```
Offset = Packet Length - Primary Header Length (24)
```

- Move read pointer.

5.4 Optional secondary header

The optional secondary header is used to provide an absolute time (i.e., clock time) stamp for data packets. The secondary header time format can be interpreted several ways. The specific interpretation is determined by the value of header flag bits 2 and 3. The structure in Figure 5-3 is used when secondary header time is to be interpreted as a Chapter 4 format value (flag bits 3−2 = 0). The structure in Figure 5-4 is used when secondary header time is to be interpreted as an IEEE-1588 format value (flag bits 3−2 = 1). The structure in Figure 5-5 is used when secondary header time is to be interpreted as an Extended Relative Time Counter (ERTC) format value (flag bits 3−2 = 2).

```
struct SuI106Ch10SecHeader_Ch4Time
    {
    uint16_t      uUnused;              //
    uint16_t      uHighBinTime;         // High order time
    uint16_t      uLowBinTime;          // Low order time
    uint16_t      uUSecs;               // Microsecond time
    uint16_t      uReserved;            //
    uint16_t      uSecChecksum;         // Secondary Header Checksum
    };
```

Figure 5-3. Optional Secondary Header Structure with IRIG 106 Ch 4 Time Representation

```
struct SuI106Ch10SecHeader_1588Time
    {
    uint32_t      uNanoSeconds;         // Nano-seconds
    uint32_t      uSeconds;             // Seconds
    uint16_t      uReserved;            //
    uint16_t      uSecChecksum;         // Secondary Header Checksum
    };
```

Figure 5-4. Optional Secondary Header Structure with IEEE-1588 Time Representation

```
struct SuI106Ch10SecHeader_ERTCTime
    {
    uint32_t      uLSLW;                // Least significant long word
    uint32_t      uMSLW;                // Most significant long word
    uint16_t      uReserved;            //
    uint16_t      uSecChecksum;         // Secondary Header Checksum
    };
```

Figure 5-5. Optional Secondary Header Structure with ERTC Time Representantion

5.5 Data payload

After the standard header and optional secondary header, each data packet begins with a CSDW. This data word provides information necessary to decode the data messages that follow. For example, it is common for the CSDW to contain a value for the number of messages that follow and to have flags that indicate what kind of intra-packet headers (IPHs) are used between messages.

Reading and decoding a data packet is accomplished by first reading the CSDW. Then read individual data messages that follow in the data packet, taking into account the appropriate IPHs and data formats. Move on to the next header and data packet when there are no more data messages to read.

When IPHs are present, they typically contain one or sometimes more than one time stamp as well as other information about the data message that follows. Commonly used structures for intra-packet time data are shown in Figure 5-6, Figure 5-7, Figure 5-8, and Figure 5-9. These four time structures will be referenced in most of the data format descriptions that follow.

```
struct SuIntrPacketTime_RTC
    {
    uint8_t       aubyRelTime[6];       // 48 bit RTC
    uint16_t      uUnused;              //
    };
```

Figure 5-6. Intra-Packet Time Stamp, 48-bit RTC

```
struct SuIntrPacketTime_Ch4Time
    {
    uint16_t      uUnused;              //
    uint16_t      uHighBinTime;         // High order time
    uint16_t      uLowBinTime;          // Low order time
    uint16_t      uUSecs;               // Microsecond time
    };
```

Figure 5-7. Intra-Packet Time Stamp, IRIG 106 Ch 4 Binary

```
struct SuIntrPacketTime_1588Time
    {
    uint32_t      uNanoSeconds;         // Nano-seconds
    uint32_t      uSeconds;             // Seconds
    };
```

Figure 5-8. Intra-Packet Time Stamp, IEEE-1588

```
struct SuIntrPacketTime_ERTCTime
    {
    uint32_t      uLSLW;                // Least significant long word
    uint32_t      uMSLW;                // Most significant long word
    };
```

Figure 5-9. Intra-Packet Time Stamp, 64-bit ERTC

5.5.1 Type 0x00, Computer-Generated Data, Format 0

Computer-Generated Data, Format 0 packets are used to store data generated internally by a recorder. The data packet begins with the CSDW shown in Figure 5-10. The data portion of the data packet is undefined and left to the discretion of the recorder manufacturer.

```
struct SuCompGen0_ChanSpec
    {
    uint32_t    uReserved;
    };
```

Figure 5-10. Type 0x00 Computer-Generated Data, Format 0 (User) CSDW

5.5.2 Type 0x01, Computer-Generated Data, Format 1 (Setup Record)

Computer-Generated Data, Format 1 packets are used to store the TMATS recorder configuration record. The data packet begins with the CSDW shown in Figure 5-11.

```
struct SuTmats_ChanSpec
    {
    uint32_t    iCh10Ver        :  8;     // Recorder Ch 10 Version
    uint32_t    bConfigChange   :  1;     // Recorder config changed
    uint32_t    bXMLFormat      :  1;     // TMATS / XML Format
    uint32_t    iReserved       : 22;     // Reserved
    };
```

Figure 5-11. Type 0x01 Computer-Generated Data, Format 1 (Setup) CSDW

Note that **`iCh10Ver`** and **`bConfigChange`** fields of the CSDW first appeared in 106-07. Since unused fields are required to be zero filled, data files prior to IRIG 106-07 will have a value of zero in the **`iCh10Ver`** field.

The **`bXMLFormat`** field first appeared in IRIG 106-09. When set the **`bXMLFormat`** flag indicates the format of the TMATS setup record is in XML format. Although the traditional TMATS format and XML TMATS formats are maintained in parallel and both are legal in all cases, the XML TMATS version is uncommon. There is currently limited support for XML-formatted TMATS.

The first data packet in a Chapter 10 data file must be a TMATS setup record. Under certain conditions, TMATS setup records may also be found later in the same recorded data file. In particular, subsequent TMATS records may occur during network data streaming of Chapter 10 data to allow network data users to learn the recorder configuration after recording and streaming has begun. The **`bConfigChanged`** flag is used to indicate whether this TMATS setup record is different than the previous TMATS setup record (i.e., the recorder configuration changed) or whether it duplicates the previous TMATS setup record.

The data that follows the CSDW in the data packet is the TMATS setup information in Chapter 9 format.

The TMATS fields change quite a bit from year to year. A summary of valid TMATS fields for each IRIG 106 release is presented in Appendix A.

The 106-09 standard made considerable changes to the TMATS "S" records, the Message Data Attributes Group. Numerous R Record fields were added to document recorder configuration and track changes to edited data files. Message filtering for PCM and 1553 are now supported. The 1553 recorder control is defined. Video time overlay is defined. Universal Asynchronous Receiver and Transmitter (UART) subchannel interface parameters are defined. Numerous "B" bus parameters were added.

The 106-11 standard added "R" group user-defined channel IDs and various event source definitions. In the "P" group numerous PCM subframe definitions were replaced. The PCM asynchronous data merge format parameters were defined. In the "D" group numerous subframe parameter definitions were deleted.

The 106-13 standard added numerous "R" group additions to define channel groups, drive groups, volumes, links, image sensor settings, and controller area network (CAN) bus subchannels. In the "P" group CRC error checking parameters and fill bits were defined. The "A" Pulse Amplitude Modulation (PAM) Attributes group was totally removed.

The 106-15 standard added a "G" group Message Digest Checksum for TMATS intregity checking. In the "R" group analog subchannel enable was defined and Fibre Channel setup was defined.

The 106-17 standard primarily added parameters to the "R" group to better define Etherenet publishing. In the "P" and "D" groups superfluous "End Frame" parameters were removed to better align usage with the XML TMATS standard.

5.5.3 Type 0x02, Computer-Generated Data, Format 2 (Recording Events)

Computer-Generated Data, Format 2 packets are used to record the occurrence of events during a recording session. Event criteria are defined in the TMATS setup record. Note that a recorded event is different and distinct from a Chapter 6 .EVENT command. A .EVENT command may result in a Recording Event packet if it has been defined in the TMATS setup record.

The layout of the CSDW is shown in Figure 5-12. The **uEventCount** field is a count of the total number of events in this packet. The **bIntraPckHdr** field indicates the presence of the optional intra-packet data header (IPDH) in the IPH.

```
struct SuEvents_ChanSpec
    {
    uint32_t    uEventCount     : 12;   // Total number of events
    uint32_t    uReserved       : 19;
    uint32_t    bIntraPckHdr    :  1;   // Intra-packet header present
    };
```

Figure 5-12. Type 0x02 Computer-Generated Data, Format 2 (Events) CSDW

There are various permutations of recorded events. In fact, there are enough permutations that it makes sense to represent the data message layout in non-specific terms. Later, during packet processing, generic data fields can be cast to their specific formats. Event data without optional data (**bIntraPckHdr = 0**) is shown in Figure 5-13. Event data with optional data (**bIntraPckHdr** = **1**) is shown in Figure 5-14. The format for the event message itself is shown in Figure 5-15.

```
struct SuEvents
    {
    uint8_t               aubyIntPktTime[8];   // Intra-packet time stamp
    struct SuEvents_Data  suData;              // Data about the event
    };
```

Figure 5-13. Type 0x02 Computer-Generated Data, Format 2 (Events) Message without Optional Data

```
struct SuEvents_with_Optional
    {
    uint8_t               aubyIntPktTime[8];   // Intra-packet time stamp
    uint64_t              suIntrPktData;       // Intra-packet data
    struct SuEvents_Data  suData;              // Data about the event
    };
```

Figure 5-14. Type 0x02 Computer-Generated Data, Format 2 (Events) Message with Optional Data

```
struct SuEvents_Data
    {
    uint32_t    uNumber         : 12;   // Event identification number
    uint32_t    uCount          : 16;   // Event count index
    uint32_t    bEventOccurence :  1;   //
    uint32_t    uReserved       :  3;   // Time tag bits
    };
```

Figure 5-15. Type 0x02 Computer-Generated Data, Format 2 (Events) Message Data

The **aubyIntPktTime** field in the data structures of Figure 5-13 and Figure 5-14 represents event time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

If the event message includes the optional **suIntrPktData** IPH field, shown in the data structure of Figure 5-14, this field holds the absolute time of the event. The format of this data is the same as the Time Data Packet Format 1, depicted in Figure 5-16 and Figure 5-17. Unfortunately Time Data Packet Format 1 represents time in more than one format and this data format does not include a method to determine which time format is used in the IPH. For this reason, this field should be used with caution, if used at all.

```
struct SuTime_MsgDayFmt
    {
    uint16_t    uTmn        :  4;      // Tens of milliseconds
    uint16_t    uHmn        :  4;      // Hundreds of milliseconds
    uint16_t    uSn         :  4;      // Units of seconds
    uint16_t    uTSn        :  3;      // Tens of seconds
    uint16_t    Reserved1   :  1;      // 0
    uint16_t    uMn         :  4;      // Units of minutes
    uint16_t    uTMn        :  3;      // Tens of minutes
    uint16_t    Reserved2   :  1;      // 0
    uint16_t    uHn         :  4;      // Units of hours
    uint16_t    uTHn        :  2;      // Tens of Hours
    uint16_t    Reserved3   :  2;      // 0
    uint16_t    uDn         :  4;      // Units of day number
    uint16_t    uTDn        :  4;      // Tens of day number
    uint16_t    uHDn        :  2;      // Hundreds of day number
    uint16_t    Reserved4   :  6;      // 0
    };
```

Figure 5-16. Type 0x11 Time Data, Format 1 Structure, Day Format

```
struct SuTime_MsgDmyFmt
    {
    uint16_t    uTmn        :  4;      // Tens of milliseconds
    uint16_t    uHmn        :  4;      // Hundreds of milliseconds
    uint16_t    uSn         :  4;      // Units of seconds
    uint16_t    uTSn        :  3;      // Tens of seconds
    uint16_t    Reserved1   :  1;      // 0
    uint16_t    uMn         :  4;      // Units of minutes
    uint16_t    uTMn        :  3;      // Tens of minutes
    uint16_t    Reserved2   :  1;      // 0
    uint16_t    uHn         :  4;      // Units of hours
    uint16_t    uTHn        :  2;      // Tens of Hours
    uint16_t    Reserved3   :  2;      // 0
    uint16_t    uDn         :  4;      // Units of day number
    uint16_t    uTDn        :  4;      // Tens of day number
    uint16_t    uOn         :  4;      // Units of month number
    uint16_t    uTOn        :  1;      // Tens of month number
    uint16_t    Reserved4   :  3;      // 0
    uint16_t    uYn         :  4;      // Units of year number
    uint16_t    uTYn        :  4;      // Tens of year number
    uint16_t    uHYn        :  4;      // Hundreds of year number
    uint16_t    uOYn        :  2;      // Thousands of year number
    uint16_t    Reserved5   :  2;      // 0
    };
```

Figure 5-17. Type 0x11 Time Data, Format 1 Structure, DMY Format

Data about the recorded event is found in the **`SuEvents_Data`** structure shown in Figure 5-15. The particular event is identified by the `uNumber` field, which corresponds to the recording event index number (i.e., the "**`n`**" value in "**`R-x\EV\ID-n`**") in the TMATS setup record for this recording. The **`uCount`** field is incremented each time this event occurs. The **`bEventOccurence`** field indicates whether the event occurred during or between record enable commands.

5.5.4 Type 0x03, Computer-Generated Data, Format 3 (Recording Index)

Computer-Generated Data, Format 3 packets record file offset values that point to various important data packets in the recording data file. Chapter 10 data files can be very large, and it's generally impractical to search for specific data without an index of some sort. Currently recording index packets are used to index the position of time packets and event packets to make it easy to move to a specific time or event in the data file; however, nothing precludes the use of index packets to index other data packet types.

Index entries are organized into a two-tier tree structure of root index packets and node index packets. A node index entry contains information (e.g., packet type, channel, offset) about the specific data packet to which it points. Multiple index entries are contained in a node index type of index packet. A root index type of index packet is used to point to node index packets in the data file. Index packets (root and node) can be stored anywhere in a data file with the exception that the final root index packet must be the last data packet in a data file. The presence of indexing is also indicated by the TMATS field "**`R-x\IDX\E`**" having a value of "**`T`**" (i.e., "true"); however, note that it is currently not unusual to find a TMATS IDX value of false, but find a valid root node packet at the end of the data file.

The layout of the CSDW is shown in Figure 5-18, and is a common format between root and node index packets. The **uIdxEntCount** field is a count of the total number of indexes in this packet. The **bIntraPckHdr** field indicates the presence of the optional IPH. The **bFileSize** field indicates the presence of the optional file size field. The **uIndexType** field indicates whether the indexes that follow are root or node indexes. If file size is present, it follows the CSDW as an unsigned 64-bit value.

```
struct SuIndex_ChanSpec
    {
    uint32_t    uIdxEntCount    : 16;   // Total number of indexes
    uint32_t    uReserved       : 13;
    uint32_t    bIntraPckHdr    :  1;   // Intra-packet header present
    uint32_t    bFileSize       :  1;   // File size present
    uint32_t    uIndexType      :  1;   // Index type
    };
```

Figure 5-18. Type 0x03 Computer-Generated Data, Format 3 (Index) CSDW

Node index packets are composed of a CSDW, an optional file size field, and multiple node index structures.

Each node index structure is composed of an intra-packet time stamp (IPTS), an optional IPH, and a node index entry data structure. The IPTS represents indexed packet data in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

If the index message includes the optional IPH field, this field holds the absolute time of the index. The format of this data is the same as the Time Data Packet Format 1, depicted in Figure 5-16 and Figure 5-17. Unfortunately Time Data Packet Format 1 represents time in more than one format and this data format does not include a method to determine which time format is used in the IPH. For this reason, this field should be used with caution, if used at all.

The structure of the node index entry is shown in Figure 5-19. The **uChannelID** field is the Channel ID of the indexed data packet. The **uDataType** field is the data type of the indexed data packet. The **uOffset** field is an unsigned eight-byte value representing the offset from the beginning of the data file to the indexed data packet. The **uOffset** field should always point to the sync pattern (0xEB25) of the indexed data packet.

```
struct SuIndex_Data
    {
    uint32_t    uChannelID      : 16;
    uint32_t    uDataType       :  8;
    uint32_t    uReserved       :  8;
    uint64_t    uOffset;
    };
```

Figure 5-19. Type 0x03 Computer-Generated Data, Format 3 (Index) Node Index Entry

Root index packets are composed of a CSDW, an optional file size field, and multiple root index entry structures. Root index structures provide information about and point to node index packets described above. The last entry is actually a pointer to the previous root index.

Each root index is composed of an IPTS, an optional IPH, and a node index data packet offset value. The IPTS represents indexed packet data time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

If the root index message includes the optional IPH field, this field holds the absolute time of the node index packet. The format of this data is the same as the Time Data Packet Format 1, depicted in Figure 5-16 and Figure 5-17. Unfortunately Time Data Packet Format 1 represents time in more than one format and this data format does not include a method to determine which time format is used in the IPH. For this reason, this field should be used with caution, if used at all.

The node index offset field of the root index packet is an eight-byte unsigned value representing the offset from the beginning of the data file to the node index packet.

The node index offset field of the last root index entry structure in a root index packet doesn't point to a node index packet. Instead this last file offset value points to the file offset value of the previous root index packet. In this way a linked list of root index packets is created. The first root index packet in the linked list is indicated by having a root index offset field value that points to itself.

5.5.5 Type 0x04, Computer-Generated Data, Format 4 (Streaming Configuration Records)

Computer-Generated Data, Format 4 packets provide information about the currently enabled network data stream or recorder configutation. Although this data packet may be found in recorded data files, it is primarily intended for use in IRIG 106 Chapter 10 standard format User Datagram Protocol (UDP) data streaming applications.

The layout of the CSDW is shown in Figure 5-20.

```
struct SuStreamingConfig_ChanSpec
    {
    uint32_t    iCh10Ver        :  8;      // Recorder Ch 10 Version
    uint32_t    iFormat         :  7;      // Config contents
    uint32_t    iLast           :  1;      // Last packet flag
    uint32_t    iReserved       : 16;      // Reserved
    };
```

Figure 5-20. Type 0x04 Computer-Generated Data, Format 4 (Streaming Configuration) CSDW

As with the Computer-Generated Data Format 1 (TMATS) packet format, the **`iCh10Ver`** field indicates the version of IRIG 106 with which the following data is compliant. Note that the Computer-Generated Data Format 4 packet uses the same version values as Computer-Generated Data Format 1 but since Computer-Generated Data Format 4 first appeared in IRIG 106-17 only values 0x0C (indicating IRIG 106-17) or greater are legal.

The **`iFormat`** field indicates the type of data to follow in the packet. Since TMATS information can be very large, Computer-Generated Data Format 4 includes the ability to segment TMATS information across multiple packets. When TMATS information is segmented the **`bLast`** flag is used to indicate the last segmented packet.

After the last TMATS segment is received, all segments can be concatenated to form a complete record. If a checksum format packet is received this value of the checksum can be compared with the computed SHA2-256 checksum of the reassembled TMATS record. The layout of the checksum format data packet is shown in Figure 5-21. Keep in mind that checksum data is in big-endian format.

```
struct SuStreamingConfig_Checksum
    {
    uint32_t    auChecksum[8];              // Checksum bits 0 to 255
    };
```

Figure 5-21. Type 0x04 Computer-Generated Data, Format 4 (Streaming Configuration) Checksum Data

When the Computer-Generated Data Format 4 packet contains currently selected channels, the data payload portion of the packet contains a list of Channel IDs. The layout of this packet format is shown in Figure 5-22.

```
struct SuStreamingConfig_Channels
    {
    uint16_t    iChannels;                  // Number of channels to follow
    uint16_t    auChannelId[1];             // Array of Channel IDs
    };
```

Figure 5-22. Type 0x04 Computer-Generated Data, Format 4 (Streaming Configuration) Channel Data

If a data channel currently has no data then data associated with this channel won't be seen. For this reason sometimes it is useful to have a list of data channels to expect in the data. The **`iChannels`** field indicates the number of channels listed. The **`auChannelId`** array is the array of 16-bit Channel IDs that are currently enabled and may be seen subsequently in the data stream.

5.5.6 Type 0x05 - 0x07, Computer-Generated Data, Format 5 – Format 7

Reserved for future use.

5.5.7 Type 0x08, PCM Data, Format 0

Reserved for future use.

5.5.8 Type 0x09, PCM Data, Format 1 (IRIG 106 Chapter 4/8)

Format 1 PCM packets are used to record PCM data frames. Most PCM data is a serial stream of bits from multiple interleaved data sources. Each data source generally operates at a different data rate and has its digital data interleaved in a fixed, defined fashion. The data from these multiplexed data sources are organized into major frames and minor frames. Format 1 PCM data records minor frames as integral units of data in packed and unpacked mode. In general the PCM data packet will contain multiple PCM minor frame data messages. There is

extensive discussion of PCM in IRIG 106 Chapter 4,[15] Chapter 8,[16] and Appendix 4.A,[17] as well as RCC Document 119, *Telemetry Applications Handbook*.[18]

A PCM minor frame is recorded in one of three major modes: unpacked, packed, and throughput. In unpacked mode, packing is disabled and each data word is padded with the number of filler bits necessary to align the first bit of each word with the next 16-bit boundary in the packet. In packed mode, packing is enabled and pad is not added to each data word; however, filler bits may be required to maintain minor frame alignment on word boundaries. In throughput mode, the PCM data are not frame synchronized so the first data bit in the packet can be any bit in the major frame. Chapter 10 discusses these modes in greater detail.

The layout of the CSDW is shown in Figure 5-23. The **uSyncOffset** field is the value of the byte offset into a major frame for the first data word in a packet, and is only valid in unpacked mode. The **bUnpackedMode**, **bPackedMode**, and **bThruMode** flags indicate unpacked mode, packed mode, and throughput mode respectively and are mutually exclusive. The **bAlignment** flag indicates 16- or 32-bit alignment of minor frames and minor frame fields. The **uMajorFrStatus** field indicates the lock status of the major frame. The **uMinorFrStatus** indicates the lock status of the minor frame. The **bMinorFrInd** flag indicates the first word of the packet is the beginning of a minor frame. The **bMajorFrInd** flag indicates the first word of the packet is the beginning of a major frame. The **bIntraPckHdr** flag indicates the presence of IPHs.

```
struct SuPcmF1_ChanSpec
    {
    uint32_t    uSyncOffset     : 18;      // Sync offset
    uint32_t    bUnpackedMode   :  1;      // Packed mode flag
    uint32_t    bPackedMode     :  1;      // Unpacked mode flag
    uint32_t    bThruMode       :  1;      // Throughput mode flag
    uint32_t    bAlignment      :  1;      // 16/32-bit alignment flag
    uint32_t    Reserved1       :  2;      //
    uint32_t    uMajorFrStatus  :  2;      // Major frame lock status
    uint32_t    uMinorFrStatus  :  2;      // Minor frame lock status
    uint32_t    bMinorFrInd     :  1;      // Minor frame indicator
    uint32_t    bMajorFrInd     :  1;      // Major frame indicator
    uint32_t    bIntraPckHdr    :  1;      // Intra-packet header flag
    uint32_t    Reserved2       :  1;      //
    };
```

Figure 5-23. Type 0x09 PCM Data, Format 1 CSDW

[15] Range Commanders Council. "Pulse Code Modulation Standards" in *Telemetry Standards*. IRIG 106-20 Chapter 4. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/chapter4.pdf.

[16] Range Commanders Council. "Digital Data Bus Acquisition Formatting Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 8. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/chapter8.pdf.

[17] Range Commanders Council. "Pulse Code Modulation Standards (Additional Information and Recommendations)" in *Telemetry Standards*. IRIG 106-20 Chapter 4 Appendix A. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/105349356/chapter4.pdf.

[18] Range Commanders Council. *Telemetry Applications Handbook*. RCC 119-06. May 2006. May be superseded by update. Retrieved 12 August 2019. Available https://www.trmc.osd.mil/wiki/download/attachments/105349396/119-06.pdf.

The optional IPH and individual PCM minor frame messages follow the CSDW. The format of the IPH is shown in Figure 5-24. The IPH, if present (indicated by **bIntraPckHdr = 1**), is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uMajorFrStatus** field indicates the lock status of the major frame. The **uMinorFrStatus** indicates the lock status of the minor frame.

```
struct SuPcmF1_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    Reserved        : 12;       //
    uint32_t    uMajorFrStatus  :  2;       // Major frame lock status
    uint32_t    uMinorFrStatus  :  2;       // Minor frame lock status
    uint32_t    Reserved        : 16;       //
    };
```

Figure 5-24. Type 0x09 PCM Data, Format 1 Intra-Packet Header

One minor frame of data follows the IPH. The length of the minor frame data is not included in the data packet. The data length must be determined from the number of bits in a minor frame specified in the TMATS parameter **P-d\MF2**. Minor frames will have padding bits added to the end to make them 16- or 32-bit aligned depending on the value of the **bAlignment** flag in the CSDW. Refer to the TMATS P-d format descriptions in Chapter 9 of the 106.

5.5.9 Type 0x0A, 0x0F PCM Data, Format 2 – Format 7

Reserved for future use.

5.5.10 Type 0x10, Time Data, Format 0

Reserved for future use.

5.5.11 Type 0x11, Time Data, Format 1 (IRIG/GPS/RTC)

Time is recorded in a data file much like any other data source. The purpose of the Time Data, Format 1 packet is to provide a correlation between an external clock source and the recorder internal 10-MHz RTC. The correlation between the RTC and clock time is described in more detail in Section 5.5.53.

The time data packet begins with the CSDW shown in Figure 5-25.

```
struct SuTimeF1_ChanSpec
    {
    uint32_t    uTimeSrc    :  4;       // Time source
    uint32_t    uTimeFmt    :  4;       // Time format
    uint32_t    bLeapYear   :  1;       // Leap year
    uint32_t    uDateFmt    :  1;       // Date format
    uint32_t    uReserved2  :  2;
    uint32_t    uReserved3  : 20;
    };
```

Figure 5-25. Type 0x11 Time Data, Format 1 CSDW

The **uTimeSrc** field indicates that the source is for the time used. Note that this field changed slightly between the 2004 and the 2005 release of Chapter 10. The **uTimeFmt** field is used to indicate the nature and type of the source of external time applied to the recorder. The **uDateFmt** field is used to determine how to interpret the time data that follows. Time representation in Day (i.e., Day of the Year) format is shown in Figure 5-16. Time representation in Day-Month-Year format is shown in Figure 5-17. The **bLeapYear** field in the CSDW is useful to convert Day of the Year to Day and Month when the year is not known.

5.5.12 Type 0x12, Time Data, Format 2 (Network Time)

Like Time Data, Format 1, the purpose of the Time Data, Format 2 packet is to provide a correlation between an external network clock source and the recorder's internal 10-MHz RTC. The correlation between the RTC and clock time is described in more detail in Section 5.5.53.

Network Time Protocol 3 (NTP3), IEEE-1588-2002 (PTP Version 1), and IEEE-1588-2008 (PTP Version 2) are currently supported.

The network time data packet begins with the CSDW shown in Figure 5-26.

```
struct SuTimeF2_ChanSpec
    {
    uint32_t    uTimeStatus :  4;      // Time status
    uint32_t    uTimeFmt    :  4;      // Network time format
    uint32_t    uReserved   : 24;
    };
```

Figure 5-26. Type 0x12 Time Data, Format 2 (Network Time) CSDW

The **uTimeStatus** field indicates the status of the time source. Currently the status can be Time Valid or Time Not Valid, although other values may be defined in the future. The **uTimeFmt** field indicates the format and interpretation of the data portion of the packet that follows.

The format of the data portion of the packet is shown in Figure 5-27.

```
struct SuTimeF2_Data
    {
    uint32_t    uSeconds;              // Integer time value
    utin32_t    uSubseconds;           // Fractional part of time value
    };
```

Figure 5-27. Type 0x12 Time Data, Format 2 Data Structure

With NTP3 and both IEEE-1588 data formats the uSeconds field represents the integer value of time. The NTP3 standard time is in Universal Coordinated Time; uses an epoch of January 1, 1900; and includes leap seconds. The IEEE-1588 standard formats are in International Atomic Time; use an epoch of January 1, 1970; and don't include leap seconds.

The fractional portion of time is in the **uSubseconds** field. The NTP standard represents subsecond time as a fraction. The most significant bit of the **uSubseconds** field represents a value of 0.5 seconds, etc. That is, there is an implied binary point between the 32-bit **uSeconds** and 32-bit **uSubseconds** field. The IEEE-1588 standard represents subsecond time in integer nano-seconds.

5.5.13 Type 0x13 - 0x17, Time Data, Format 3 – Format 7

Reserved for future use.

5.5.14 Type 0x18, Military Standard (MIL-STD) 1553 Data, Format 0

Reserved for future use.

5.5.15 Type 0x19, MIL-STD-1553 Data, Format 1 (MIL-STD-1553B Data)

Format 1 MIL-STD-1553 data packets are used to record the MIL-STD-1553 message transactions on a bus. In general the 1553 data packet will contain multiple 1553 messages.

The layout of the CSDW is shown in Figure 5-28. The **`uMsgCnt`** field indicates the number of messages contained in the data packet. The **`uTTB`** field indicates the 1553 message bit to which the time tag corresponds.

```
struct Su1553F1_ChanSpec
    {
    uint32_t    uMsgCnt      : 24;      // Message count
    uint32_t    Reserved     :  6;
    uint32_t    uTTB         :  2;      // Time tag bits
    };
```

Figure 5-28. Type 0x19 MIL-STD-1553 Data, Format 1 CSDW

The individual 1553 messages follow the CSDW. Each 1553 message has an IPTS, an IPDH data word, and then the actual 1553 message. The layout of the message header is shown in Figure 5-29. The **`aubyIntPktTime`** field is an eight-byte value. The specific interpretation of this field is determined by packet header flags. This time is in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. Various bit flags and values are found in the IPH.

```
struct Su1553F1_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint16_t    Reserved1       : 3;    // Reserved
    uint16_t    bWordError      : 1;
    uint16_t    bSyncError      : 1;
    uint16_t    bWordCntError   : 1;
    uint16_t    Reserved2       : 3;
    uint16_t    bRespTimeout    : 1;
    uint16_t    bFormatError    : 1;
    uint16_t    bRT2RT          : 1;
    uint16_t    bMsgError       : 1;
    uint16_t    iBusID          : 1;
    uint16_t    Reserved3       : 2;
    uint8_t     uGapTime1;
    uint8_t     uGapTime2;
    uint16_t    uMsgLen;
    };
```

Figure 5-29. Type 0x19 MIL-STD-1553 Data, Format 1 Intra-Packet Header

The amount of data that follows the IPH is variable. The data length in bytes is given in the **uMsgLen** field and is necessary to determine the amount of additional data to read to complete the message.

The layout and order of 1553 command word(s), status word(s), and data word(s) in the recorded 1553 message is not fixed but rather is the same as it would be found "on the wire". It's not therefore possible to define a fixed data structure representation for the message data. The first word in the data will always be a command word, but it will be necessary to use the command word as well as the **bRT2RT** flag to determine the offsets of the other message structures such as the status and data word(s). The layouts of the various types of 1553 messages are shown in Figure 5-30. When calculating data word count, be careful to take Mode Codes and word count wrap around into account. An algorithm to determine the number of data words in a message is shown in C-like pseudo-code in Figure 5-31.

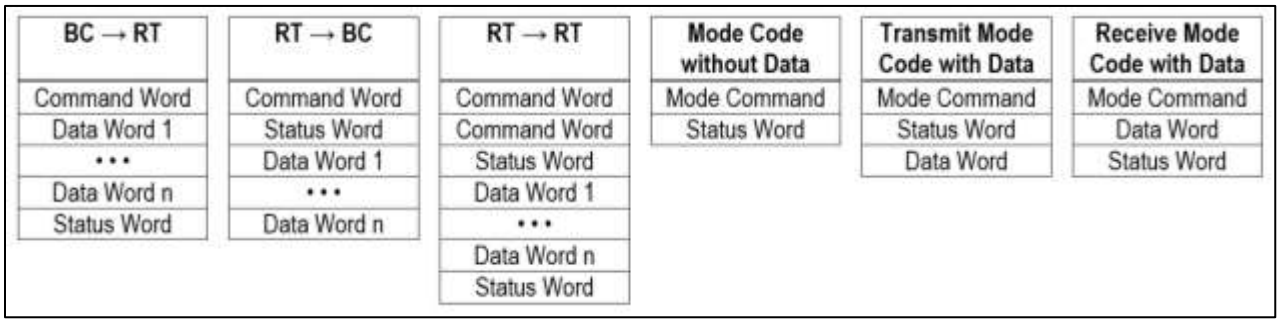

Figure 5-30. 1553 Message Word Layout

```
// Mode Code case
if (Subaddress = 0x0000) or (Subaddress = 0x001f)
    if (WordCount & 0x0010) DataWordCount = 1
    else                    DataWordCount = 0

// Non-Mode Code case
else
    if (WordCount = 0)      DataWordCount = 32
    else                    DataWordCount = WordCount
```

Figure 5-31. Algorithm to Determine 1553 Data Word Count

5.5.16 Type 0x1A, MIL-STD-1553 Data, Format 2 (16PP194 Bus)

Format 2 of MIL-STD-1553 packets is used to record data from the 16PP194 data bus. The 16PP194 data bus is used as the F-16 weapons multiplex bus. It is defined in document 16PP362A Weapons MUX (WMUX) Protocol.[19] A 16PP194 transaction consists of six 32-bit words consisting of a 16PP194 Command, Command Echo, Response, GO/NOGO, GO/NOGO Echo, and Status as illustrated in Figure 5-32. Multiple transactions may be encoded into the data portion of a single packet.

[19] Lockheed Martin Corporation. "Advanced Weapons Multiplex Data Bus." 8 June 2010. May be superseded by update. Retrieved 3 June 2015. Available to RCC members with Private Portal access at https://wsdmext.wsmr.army.mil/site/rccpri/Limited_Distribution_References/16PP362B.pdf.

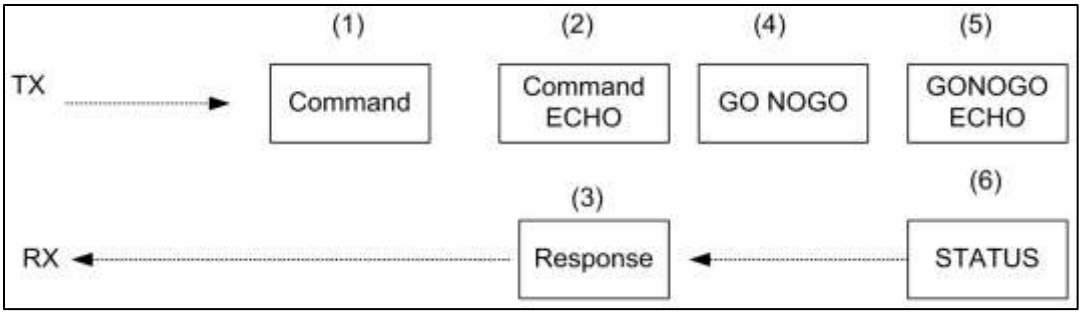

Figure 5-32. 16PP194 Message Transaction

The layout of the CSDW is show in Figure 5-33. The 16PP194 packet can contain multiple bus transactions. The **uMsgCnt** field indicates the number of 16PP194 messages in the packet.

```
struct Su1553F2_ChanSpec
    {
    uint32_t    uMsgCnt;                // Message count
    };
```

Figure 5-33. Type 0x1A MIL-STD-1553 Data, Format 2 (16PP194) CSDW

The 16PP194 message word is 26 bits in length and consists of 16 data bits, 4 address bits, 4 sub-address bits, a parity bit, and a sync bit. Only the 24 bits of data, address, and sub-address values are mapped into the 16PP194 recorded data word. Sync and parity bits are not recorded. The mapping of these bits is shown in Figure 5-34.

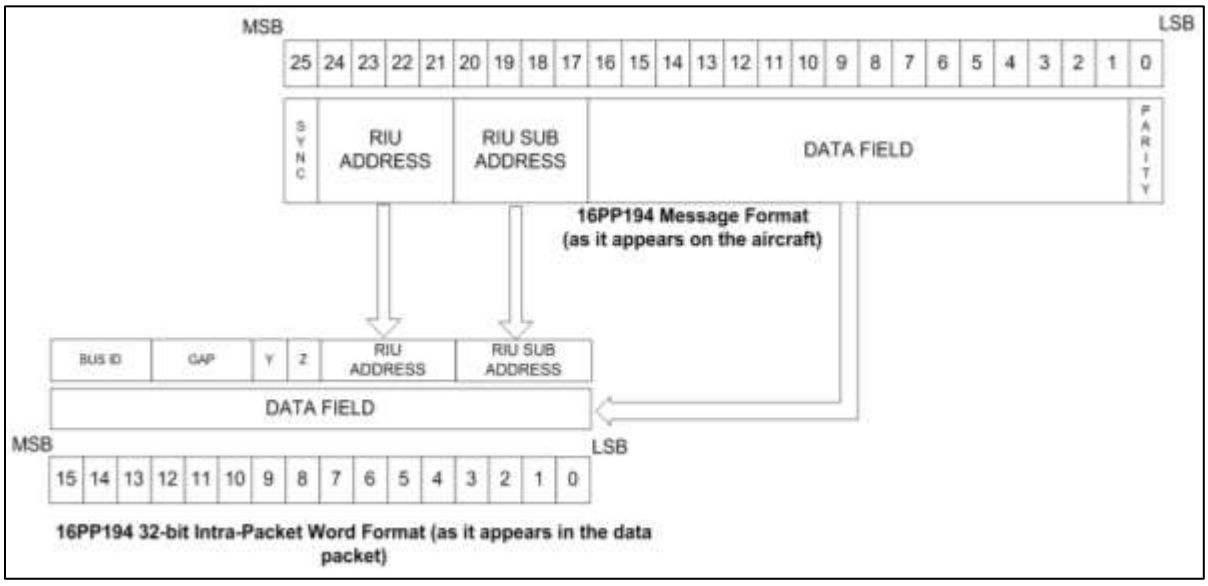

Figure 5-34. 16PP194 to IRIG 106 Chapter 10 Data Bit Mapping

The layout of the recorded 16PP194 word is shown in Figure 5-35. The **uDataWord** field contains the message data. The **uRiuSubAddr** field is the remote interface unit (RIU) sub-address. The **uRiuAddr** field is the RIU address. The **bParityError** flag indicates a parity error occurred during reception of the message. The **bWordError** flag indicates a Manchester decoding error occurred during reception of the message. The **uGap** field indicates

the general range of gap time. The mapping of **uGap** values to gap time ranges can be found in the Chapter 10 standard. The **uBusID** field indicates the bus on which the message occurred. Bus identification can be found in the Chapter 10 standard. The layout of a complete 16PP194 transaction is shown in Figure 5-36.

```
struct Su16PP194_DataWord
    {
    uint16_t    uDataWordHigh   : 8;    // Data word bits 24-17
    uint16_t    bParityError    : 1;    // Parity error
    uint16_t    bWordError      : 1;    // Word bit error
    uint16_t    uGap            : 3;    // Gap time
    uint16_t    uBusID          : 3;    // Bus ID
    uint16_t    uDataWordLow;           // Data word bits 1-16
    };
```

Figure 5-35. 16PP194 Word Layout

```
struct Su16PP194_Transaction
    {
    struct Su16PP194_Word  suCommand;
    struct Su16PP194_Word  suResponse;
    struct Su16PP194_Word  suCommandEcho;
    struct Su16PP194_Word  suNoGo;
    struct Su16PP194_Word  suNoGoEcho;
    struct Su16PP194_Word  suStatus;
    };
```

Figure 5-36. 16PP194 Transaction Layout

5.5.17 Type 0x1B - 0x1F, MIL-STD-1553 Data, Format 3 - Format 7

Reserved for future use.

5.5.18 Type 0x20, Analog Data, Format 0

Reserved for future use.

5.5.19 Type 0x21, Analog Data, Format 1 (Analog Data)

Analog Data, Format 1 packets are used to record digitized analog signals. In general the analog data packet will contain multiple digitized values from multiple analog channels. Digitized signal values can be stored in either a unpacked or packed fashion. Unpacked storage is where each sample occupies one or more 16-bit words with unused bits zero-filled. Packed storage concatenates samples to use the least amount of storage possible, but samples will straddle 16-bit word boundaries.

The layout of the CSDW is shown in Figure 5-37. Analog packets may have one or multiple CSDWs. If all analog subchannels are the same then the **bSame** value will equal "**1**" and only one CSDW will be present. If subchannels are configured differently then there will be one CSDW for each subchannel. The **uMode** field indicates whether the samples are stored in packed or unpacked mode and how unused bits are zero-filled. The **uLength** field indicates the number of bits in the digitized sample. The **uSubChan** field indicates the number of the current subchannel. The **uTotChan** field indicates the total number of subchannels in the packet. The **uFactor** field indicates the exponent of the power of 2 as a sampling rate factor denominator.

```
struct SuAnalogF1_ChanSpec
    {
    uint32_t    uMode           :  2;       // Packed or Unpacked
    uint32_t    uLength         :  6;       // Bits in A/D value
    uint32_t    uSubChan        :  8;       // Subchannel number
    uint32_t    uTotChan        :  8;       // Total number of subchannels
    uint32_t    uFactor         :  4;       // Sample rate exponent
    uint32_t    bSame           :  1;       // One/multiple CSDW
    uint32_t    iReserved       :  3;       //
    };
```

Figure 5-37. Type 0x21 Analog Data, Format 1 CSDW

Sample rate, least significant bit value, offset, and other analog parameters are recorded in the TMATS packet. The layout of the sequential samples is described in detail in the Chapter 10 standard.

5.5.20 Type 0x22 - 0x27, Analog Data, Format 2 - Format 7

Reserved for future use.

5.5.21 Type 0x28, Discrete Data, Format 0

Reserved for future use.

5.5.22 Type 0x29, Discrete Data, Format 1 (Discrete Data)

Discrete Data, Format 1 packets are used to record the state of discrete digital signals. In general the discrete data packet will contain multiple values from multiple discrete events.

The layout of the CSDW is shown in Figure 5-38. The **`uRecordState`** and **`uAlignment`** flags are components of the discrete packet mode field. See the Chapter 10 standard for further details. The **`uLength`** value is the number of discrete bits that follow in the packet.

```
struct SuDiscreteF1_ChanSpec
    {
    uint32_t    uRecordState    :  1;       // Record on state/time
    uint32_t    uAlignment      :  1;       // Data alignment
    uint32_t    uReserved1      :  1;       //
    uint32_t    uLength         :  5;       // Number of bits
    uint32_t    uReserved2      : 24;       //
    };
```

Figure 5-38. Type 0x29 Discrete Data, Format 1 CSDW

The layout of the Discrete Data message is shown in Figure 5-39. Each message contains a time stamp and the state of the discrete data input signals. The **`aubyIntPktTime`** field in the data structures represents event time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. Note that a **`uLength`** value of zero indicates an actual length of 32 bits.

```
struct SuDiscreteF1
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    suData;                     // Data about the event
    };
```

Figure 5-39. Type 0x29 Discrete Data, Format 1 Message

Versions of Chapter 10 prior to the 2009 release incorrectly stated that bit 7 of the packet flags (in the packet header) is used to determine if the intra-packet time is relative time or absolute time. The correct bit to use is bit 6.

5.5.23 Type 0x2A - 0x2F, Discrete Data, Format 2 - Format 7

Reserved for future use.

5.5.24 Type 0x30, Message Data, Format 0 (Generic Message Data)

Message Data packets are used to record data from sources that do not have a defined packet type in the Chapter 10 standard. Examples of this might be the H-009 bus found on older F-15 aircraft or the high-speed data bus found on older F-16 aircraft. The Chapter 10 standard implies that Message Data packets represent "serial" communications data. In practice, there are few restrictions on the content or source of the data for Message Data packets.

Message Data packets do not contain any field to indicate what format or type of data they contain or how to interpret the data contents. Message Data packets are distinguished by their Channel ID and Subchannel values. The TMATS setup provides fields specifying a subchannel name (**R-x\MCNM-n-m**) for each subchannel number (**R-x\MSCN-m-n**) and Channel ID (**R-x\TK1-m**). Use the subchannel name to determine how to decode each Channel ID/subchannel combination. There currently is no standard for subchannel names and no registry of "well-known" names.

Individual Message Data messages are restricted to no more than 65,535 bytes (64 kilobytes [kb]). A Message Data packet can contain multiple data messages, a single data message, or a segment of a large (>64 kb) data message. A Message Data packet consists of a CSDW followed by one or more messages. The layout of the CSDW is shown in Figure 5-40. The **uType** value indicates whether the data is a complete message or a segment of a large message. The **uCounter** value indicates either the number of data packets to follow or, for the segmented case, the segment number of the large data packet segment that follows.

```
struct SuMessageF0_ChanSpec
    {
    uint32_t    uCounter        : 16;       // Message/segment counter
    uint32_t    uType           :  2;       // Complete/segment type
    uint32_t    uReserved       : 14;
    };
```

Figure 5-40. Type 0x30 Message Data, Format 0 CSDW

The layout of the Message Data message IPH is shown in Figure 5-41. Each header contains a time stamp and some additional information about the data that follows in the message. The **aubyIntPktTime** field in the intra-packet data structure represents event time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended

relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uMsgLength** value indicates the number of data bytes in the message. The **uSubChannel** value identifies the specific subchannel from which this data came. The message data immediately follows the IPH. Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data. The **bFmtError** and **bDataError** flags indicate that errors have occurred in the reception of the data. The recorded data may, therefore, be invalid and unusable.

```
struct MessageF0_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uMsgLength      : 16;       // Message length
    uint32_t    uSubChannel     : 14;       // Subchannel number
    uint32_t    bFmtError       :  1;       // Format error flag
    uint32_t    bDataError      :  1;       // Data error flag
    };
```

Figure 5-41. Type 0x30 Message Data, Format 0 Intra-Packet Header

5.5.25 Type 0x31 - 0x37, Message Data, Format 1 - Format 7

Reserved for future use.

5.5.26 Type 0x38, ARINC 429 Data, Format 0 (ARINC429 Data)

Format 0 ARINC 429 data packets are used to record data messages from an ARINC 429 data bus. The ARINC 429 standard[20] defines a unidirectional data bus commonly found on commercial and transport aircraft. Words are 32 bits in length and most messages consist of a single data word. Messages are transmitted at either 12.5 or 100 kbit/s from a transmitting system to one or more receiving systems. The transmitter is always transmitting either 32-bit data words or the NULL state. An ARINC data packet can contain multiple messages.

The layout of the CSDW is shown in Figure 5-42. The **uMsgCount** field indicates the number of recorded ARINC 429 messages.

```
struct SuArinc429F0_ChanSpec
    {
    uint32_t    uMsgCount       : 16;       // Message count
    uint32_t    Reserved        : 16;       //
    };
```

Figure 5-42. Type 0x60 ARINC 429 Data, Format 0 CSDW

Individual ARINC 429 data messages follow the CSDW. Each message is preceded with an IPDH followed by the ARINC 429 data word.

The layout of the ARINC 429 data message IPDH is shown in Figure 5-43. The **uGapTime** field is the time between the beginning of the preceding bus word and the beginning of the current bus word in 0.1-microsecond increments. The **uBusSpeed** field indicates the bit rate of the recorded bus message. The **bParityError** flag indicates the presence of a parity

[20] Aeronautical Radio, Inc. *Mark 33 Digital Information Transfer System (DITS)*. ARINC 429. Annapolis: ARINC, 1995.

data error. The **bFormatError** flag indicates the presence of one of several types of data format errors. The **uBusNum** field identifies the specific ARINC 429 bus associates with the recorded data message.

```
struct SuArinc429F0_Header
    {
    uint32_t    uGapTime        : 20;      // Gap Time
    uint32_t    Reserved        :  1;      //
    uint32_t    uBusSpeed       :  1;      // Bus Speed
    uint32_t    bParityError    :  1;      // Parity Error
    uint32_t    bFormatError    :  1;      // Data type
    uint32_t    uBusNum         :  8;      // Bus number
    };
```

Figure 5-43. Type 0x38 ARINC 429 Data, Format 0 Intra-Packet Data Header

The layout of the individual ARINC 429 data words is shown in Figure 5-44. Refer to the ARINC 429 standard for the interpretation of the specific data fields.

```
struct SuArinc429F0_Data
    {
    uint32_t    uLabel          :  8;      // Label
    uint32_t    uSDI            :  2;      // Source/Destination ID
    uint32_t    uData           : 19;      // Data
    uint32_t    uSSM            :  2;      // Sign/Status Matrix
    uint32_t    uParity         :  1;      // Parity
    };
```

Figure 5-44. Type 0x38 ARINC 429 Data Format

Note that the value in the **uLabel** field is stored as it is encountered on the bus, resulting in the bits being in reverse order. To correctly interpret the **uLabel** field it is necessary to reverse the bit ordering. Note also that when formatting the **uLabel** value for output the value is commonly formatted in octal format.

5.5.27 Type 0x39 - 0x3F, ARINC 429 Data, Format 1 - Format 7

Reserved for future use.

5.5.28 Type 0x40, Video Data, Format 0 (MPEG-2/H.264 Video)

Video Data, Format 0 packets are used to record digitized video and associated audio signals. Format 0 packets are restricted to contain only MPEG-2 transport stream (TS) packets. Video can be encoded with either MPEG-2 Main Profile Main Level encoding or H.264 (also known as MPEG-4 Part 10 and MPEG-4 Advanced Video Coding [AVC]) Main Profile Level 3 encoding. The H.264 standard[21] is usually the preferred encoder for lower bit rate video. This encoding is usually referred to as Standard Definition and has a maximum resolution of 720 by 576 pixels but is frequently less.

The layout of the CSDW is shown in Figure 5-45. The **uType** value indicates the specific video encoding type in the MPEG-2 stream. This field was first defined in IRIG 106-07

[21] International Telecommunications Union Telecommunication Standardization Sector. "Advanced Video Coding for Generic Audiovisual Services." ITU-T H.264. June 2019. May be superseded by update. Retrieved 13 August 2019. Available at https://www.itu.int/rec/T-REC-H.264.

(Data Version 0x03). It was reserved and zero filled in previous versions of Chapter 10. The **bKLV** flag indicates the presence of key-length-value (KLV) metadata fields in the video data. The **bSRS** flag indicates whether or not the embedded video clock is synchronized with the RTC. The **bIntraPckHdr** flag indicates the presence of IPH data in each video data message. The **bET** flag indicates the presence of embedded time in the video data.

```
struct SuVideoF0_ChanSpec
    {
    uint32_t    Reserved     : 24;
    uint32_t    uType        :  4;      // Payload type
    uint32_t    bKLV         :  1;      // KLV present
    uint32_t    bSRS         :  1;      // SCR/RTC Sync
    uint32_t    bIntraPckHdr :  1;      // Intra-Packet Header
    uint32_t    bET          :  1;      // Embedded Time
    };
```

Figure 5-45. Type 0x40 Video Data, Format 0 CSDW

All TS packets follow the CSDW. All TS packets are a fixed size of 188 bytes. If the `bIntraPckHdr` flag is set, each TS packet will be preceded with an eight-byte IPTS. The intra-packet time represents TS time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. Format 0 does not include a packet count. Instead the number of TS packets can be calculated from the size of the data packet found in the Chapter 10 header.

Format 0 video data can be readily decoded with commonly available MPEG libraries such as the open-source ffmpeg library. A 188-byte TS packet is best thought of as a contiguous stream of 1504 bits. All TS packets are stored in Format 0 packets as a series of 16-bit words. The first Format 0 data word holds the first 16 TS packet bits. The first TS packet, or bit 0, is the left-most bit (most significant bit) in the first Format 0 packet word. Note that the description of this bit-ordering alignment in a Format 0 packet has been frequently depicted wrong in the various IRIG 106 Chapter 10 releases. Most MPEG decoder libraries such as ffmpeg commonly take as input a 188-byte array of TS data. Due to the use of 16-bit words to store TS data in Format 0 packets, TS data needs to be byte-swapped as it is read from a Chapter 10 data file and put into a buffer for decoding by a software library expecting byte-aligned TS data.

5.5.29 Type 0x41, Video Data, Format 1 (ISO 13818-1 MPEG-2)

Video Data, Format 1 packets are used to record digitized video and associated audio signals. Format 1 packets can support the complete MPEG-2 ISO/IEC 13818-1:2019[22] standard for both program streams (PSs) and TSs. Any profile and level combination can be used in Format 1.

The layout of the CSDW is shown in Figure 5-46. The **uPacketCnt** value is the number of MPEG-2 packets in the data packet. The **uType** value indicates whether the video packet is a PS or TS. The **uMode** value indicates whether the video packet uses constant or

[22] ISO/IEC. *Information Technology – Generic Coding of Moving Pictures and Associated Audio Information: Systems*. ISO/IEC 13818-1:2019. June 2019. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/75928.html.

variable rate encoding. The **bET** flag indicates the presence of embedded time in the video data. The **uEPL** value indicates the video packet encoding profile and level used. The **bIntraPckHdr** flag indicates the presence of IPH data in each video data message. The **bSRS** flag indicates whether or not the embedded video clock is synchronized with the RTC. The **bKLV** flag indicates the presence of KLV metadata fields in the video data.

```
struct SuVideoF1_ChanSpec
    {
    uint32_t    uPacketCnt   : 12;      // Number of packets
    uint32_t    uType        :  1;      // TS/PS type
    uint32_t    uMode        :  1;      // Const/Var mode
    uint32_t    bET          :  1;      // Embedded Time
    uint32_t    uEPL         :  4;      // Encoding Profile and Level
    uint32_t    bIntraPckHdr :  1;      // Intra-Packet Header
    uint32_t    bSRS         :  1;      // SCR/RTC Sync
    uint32_t    bKLV         :  1;      // KLV present
    uint32_t    uReserved    : 10;
    };
```

Figure 5-46. Type 0x40 Video Data, Format 1 CSDW

The CSDW is followed by MPEG-2 PS or TS packets. All TS packets are a fixed length of 188 bytes, but PS packets are variable length. If the **bIntraPckHdr** flag is set, each MPEG-2 packet will be preceded with an eight-byte IPTS. The intra-packet time represents MPEG packet time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

Format 1 does not include a method to separate individual MPEG packets from within the Format 1 packet other than determining the MPEG packet size from the MPEG packet data. Determining MPEG packet size is fairly complicated and involves quite a bit of knowledge about MPEG internal data structures. For this reason, the use of IPHs between MPEG packets should be carefully considered. It could make decoding Format 1 packets quite complicated.

5.5.30 Type 0x42, Video Data, Format 2 (ISO 14496 MPEG-4 Part 10 AVC/H.264)

Video Data, Format 2 packets are used to record digitized video and associated audio signals. Format 2 packets can support the complete MPEG-2 ISO/IEC 13818-1:2019 standard for both PSs and TSs, and provides all H.264 (also known as MPEG-4 Part 10 and MPEG-4 AVC) encoding levels and profiles.

The layout of the CSDW is shown in Figure 5-47. The **uPacketCnt** value is the number of MPEG-2 packets in the data packet. The **uType** value indicates whether the video packet is a PS or TS. The **uMode** value indicates whether the video packet uses constant or variable rate encoding. The **bET** flag indicates the presence of embedded time in the video data. The **uEP** value indicates the video packet encoding profile used. The **bIntraPckHdr** flag indicates the presence of IPH data in each video data message. The **bSRS** flag indicates whether or not the embedded video clock is synchronized with the RTC. The **bKLV** flag indicates the presence of KLV metadata fields in the video data. The **uEL** value indicates the video packet encoding profile and level used. The **uAET** field indicates the type of AVC/H.264 audio encoding used.

```
struct SuVideoF2_ChanSpec
    {
    uint32_t    uPacketCnt   : 12;      // Number of packets
    uint32_t    uType        :  1;      // TS/PS type
    uint32_t    uMode        :  1;      // Const/Var mode
    uint32_t    bET          :  1;      // Embedded Time
    uint32_t    uEP          :  4;      // Encoding Profile
    uint32_t    bIntraPckHdr :  1;      // Intra-Packet Header
    uint32_t    bSRS         :  1;      // SCR/RTC Sync
    uint32_t    bKLV         :  1;      // KLV present
    uint32_t    uEL          :  4;      // Encoding Level
    uint32_t    uAET         :  1;      // Audio Encoding Type
    uint32_t    uReserved    :  5;
    };
```

Figure 5-47. Type 0x40 Video Data, Format 2 CSDW

The CSDW is followed by MPEG-2 PS or TS packets. All TS packets are a fixed length of 188 bytes, but PS packets are variable length. If the **`bIntraPckHdr`** flag is set, each MPEG-2 packet will be preceded with an eight-byte IPTS. The intra-packet time represents MPEG packet time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

Format 2 does not include a method to separate individual MPEG packets from within the Format 2 packet other than determining the MPEG packet size from the MPEG packet data. Determining MPEG packet size is fairly complicated and involves quite a bit of knowledge about MPEG internal data structures. For this reason, the use of IPHs between MPEG packets should be carefully considered. It could make decoding Format 2 packets quite complicated.

5.5.31 Type 0x43, Video Data, Format 3 (MJPEG)

Video Data, Format 3 packets are used to record digitized video as a series of individual JPEG images. Unlike most other video formats MJPEG doesn't have provisions for encoding audio. Video encoding is in accordance with (IAW) ISO/IEC 10918 Part 1[23].

The layout of the CSDW is shown in Figure 5-48. The **`bIntraPckHdr`** flag indicates the presence of IPH data in each video data message. The **`uSum`** field indicates whether the data packet contains a data segment, one complete frame, or multiple complete frames. The **`uPart`** field indicates the type of data segment for segmented video packets.

```
struct SuVideoF3_ChanSpec
    {
    uint32_t    uReserved    : 27;
    uint32_t    bIntraPckHdr :  1;      // Intra-Packet Header
    uint32_t    uSum         :  2;      // Packet segmentation
    uint32_t    uPart        :  2;      // Segmented packet part
    };
```

Figure 5-48. Type 0x43 Video Data, Format 3 (MJPEG) CSDW

[23] ISO/IEC. "General sequential and progressive syntax", Annex B, section B.2, in *Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines*. ISO/IEC 10918-1:1994. May be superseded by update. Geneva: International Organization for Standardization, 1994.

The CSDW is followed by MJPEG packets. If the **bIntraPckHdr** flag is set, each MJPEG packet will be preceded with an IPH as shown in Figure 5-49. The intra-packet time represents MJPEG packet time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.. The **uFrameLen** field holds the size in bytes of the MJPEG frame to follow.

```
struct SuVideoF3_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint32_t    uFrameLen;              // Message data length
    };
```

Figure 5-49. Type 0x43 Video Data, Format 3 (MJPEG) Intra-packet Header

Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data.

5.5.32 Type 0x44, Video Data, Format 4 (MJPEG-2000)

Video Data, Format 4 packets are used to record digitized video as a series of individual JPEG-2000 images. Video encoding is IAW ISO/IEC 15444-3:2007 Motion JPEG 2000[24].

The layout of the CSDW is shown in Figure 5-50. The **bIntraPckHdr** flag indicates the presence of IPH data in each video data message. The **uSum** field indicates whether the data packet contains a data segment, one complete frame, or multiple complete frames. The **uPart** field indicates the type of data segment for segmented video packets.

```
struct SuVideoF4_ChanSpec
    {
    uint32_t    uReserved    : 27;
    uint32_t    bIntraPckHdr :  1;      // Intra-Packet Header
    uint32_t    uSum         :  2;      // Packet segmentation
    uint32_t    uPart        :  2;      // Segmented packet part
    };
```

Figure 5-50. Type 0x44 Video Data, Format 3 (MJPEG-2000) CSDW

The CSDW is followed by MJPEG packets. If the **bIntraPckHdr** flag is set, each MJPEG-2000 packet will be preceded with an IPH as shown in Figure 5-51. The intra-packet time represents MJPEG-2000 packet time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uFrameLen** field holds the size in bytes of the MJPEG-2000 frame to follow.

[24] ISO/IEC. *Information technology: JPEG 2000 image coding system: motion JPEG 2000*. ISO/IEC 15444-3:2007. Geneva: International Organization for Standardization, 2007.

```
struct SuVideoF4_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint32_t    uFrameLen;              // Message data length
    };
```

Figure 5-51. Type 0x44 Video Data, Format 4 (MJPEG-2000) Intra-packet Header

Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data.

5.5.33 Type 0x45 - 0x47, Video Data, Format 5 - Format 7

Reserved for future use.

5.5.34 Type 0x48, Image Data, Format 0 (Image Data)

Image Data, Format 0 packets are used to record digitized video images.

The layout of the CSDW is shown in Figure 5-52. The **uLength** value is the number of bytes in each segment. The **bIntraPckHdr** flag indicates the presence of the IPH. The **uSum** value indicates if and how an image is segmented in the data packet. The **uPart** value indicates which part of a possibly segmented image is contained in the data packet.

```
struct SuImageF0_ChanSpec
    {
    uint32_t    uLength         : 27;       // Segment byte length
    uint32_t    bIPH            :  1;       // Intra-packet header flag
    uint32_t    uSum            :  2;       // Packet segmentation
    uint32_t    uPart           :  2;       //
    };
```

Figure 5-52. Type 0x48 Image Data, Format 0 CSDW

Individual image data messages follow the CSDW. Each message may have an optional IPH followed by the image data. The IPH, if present (indicated by **bIntraPckHdr** = **1**), is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

The format of the image data portion of the packet is not specified in Chapter 10. The image format is specified in the TMATS setup record attribute "**R-x\SIT-n**". Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data.

5.5.35 Type 0x49, Image Data, Format 1 (Still Imagery)

Image Data, Format 1 packets are used to record digitized still images. Several formats for image compression are supported by Format 1.

The layout of the CSDW is shown in Figure 5-53. The **uFormat** value indicates the format of the image data. The **bIntraPckHdr** flag indicates the presence of the IPH. The

uSum value indicates if and how an image is segmented in the data packet. The **uPart** value indicates which part of a possibly segmented image is contained in the data packet.

```
struct SuImageF1_ChanSpec
    {
    uint32_t    uReserved       : 23;       // Reserved
    uint32_t    uFormat         :  4;       // Image format
    uint32_t    bIPH            :  1;       // Intra-packet header flag
    uint32_t    uSum            :  2;       // Packet segmentation
    uint32_t    uPart           :  2;       // Segmented packet part
    };
```

Figure 5-53. Type 0x49 Image Data, Format 1 CSDW

Individual image data messages follow the CSDW. Each message may have an optional IPH (indicated by **bIntraPckHdr** = **1**) followed by the image data. The format of the IPH is shown in Figure 5-54. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uMsgLength** value indicates the length of the following still image or segment.

```
struct SuImageF1_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uMsgLength;                 // Message length
    };
```

Figure 5-54. Type 0x49 Image Data, Format 1 Intra-Packet Header

The format of the image data portion of the packet is not specified in Chapter 10. The image format is specified in the TMATS setup record attribute "**R-x\SIT-n**". Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data.

5.5.36 Type 0x4A, Image Data, Format 2 (Dynamic Imagery)

Image Data, Format 2 packets are used to record digitized dynamic images. The main feature that distinguishes this format from Image Data, Formats 0 and 1 is that it has an update rate associated with it. Image frame rate is specified in TMATS setup record attribute "R-x\IFR-n". This format also supports a much wider range of image format types. The image format is also stored on a per-image basis in the IPH.

The layout of the CSDW is shown in Figure 5-55. The **uFormat** value indicates the format of the image data. The **bIntraPckHdr** flag indicates the presence of the IPH. The **uSum** value indicates if and how an image is segmented in the data packet. The **uPart** value indicates which part of a possibly segmented image is contained in the data packet.

```
struct SuImageF2_ChanSpec
    {
    uint32_t    uReserved       : 21;       // Reserved
    uint32_t    uFormat         :  6;       // Image format
    uint32_t    bIPH            :  1;       // Intra-packet header flag
    uint32_t    uSum            :  2;       // Packet segmentation
    uint32_t    uPart           :  2;       // Segmented packet part
    };
```

Figure 5-55. Type 0x4A Image Data, Format 2 (Dynamic Imagery) CSDW

Individual image data messages follow the CSDW. Each message may have an optional IPH (indicated by **`bIntraPckHdr`** = 1) followed by the image data. The format of the IPH is shown in Figure 5-56. The **`aubyIntPktTime`** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **`uMsgLength`** value indicates the length of the following still image or segment.

```
struct SuImageF2_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uMsgLength;                 // Message length
    };
```

Figure 5-56. Type 0x4A Image Data, Format 2 (Dynamic Imagery) Intra-packet Header

Note that an even number of bytes is allocated for message data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data.

5.5.37 Type 0x4B - 0x4F, Image Data, Format 3 - Format 7

Reserved for future use.

5.5.38 Type 0x50, UART Data, Format 0

Format 0 UART packets are used to record character data from an asynchronous serial interface. In general the UART data packet will contain multiple buffers of serial data. The Chapter 10 standard has no provisions for specifying how characters will be grouped into blocks of data other than the 100-millisecond maximum data block time. It is common for recorder vendors to provide a mechanism for determining serial message boundaries and recording a complete serial message into a single UART data message based on detecting "dead time" between messages.

The layout of the CSDW is shown in Figure 5-57. The **`bIntraPckHdr`** flag is used to indicate if an optional IPTS is included with each UART data message.

```
struct SuUartF0_ChanSpec
    {
    uint32_t    Reserved     : 31;
    uint32_t    bIPH         :  1;      // Intra-Packet Header enabled
    };
```

Figure 5-57. Type 0x50 UART Data, Format 0 CSDW

The UART Data message may have an optional IPTS. If so, this time field will be an eight-byte value. Time is represented in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

The layout of the UART Data message IPDH is shown in Figure 5-58. The **uDataLength** value indicates the number of data bytes in the message. The **uSubChannel** value identifies the specific subchannel this data came from. The message data immediately follows the IPDH. Note that an even number of bytes is allocated for UART data. If the data contains an odd number of bytes, then one unused filler byte is inserted at the end of the data. The **bParityError** flag indicate that errors have occurred in the reception of the data. The recorded data may, therefore, be invalid and unusable.

```
struct SuUartF0_Header
    {
    uint16_t    uDataLength     : 16;    // Length in bytes
    uint16_t    uSubchannel     : 14;    // Subchannel
    uint16_t    uReserved       : 1;
    uint16_t    bParityError    : 1;     //Parity Error
    };
```

Figure 5-58. Type 0x40 UART Data, Format 0 Intra-Packet Data Header

5.5.39 Type 0x51 - 0x57, UART Data, Format 1 - Format 7

Reserved for future use.

5.5.40 Type 0x58, IEEE-1394 Data, Format 0 (IEEE-1394 Transaction)

Format 0 IEEE-1394 data packets are used to record data messages at the transaction layer of the IEEE 1394 serial data bus. Currently IEEE 1394-1995, IEEE 1394a, and IEEE 1394b are supported.

There are two major classes of IEEE 1394 data transfer: synchronous and isochronous. Synchronous data transfer is used to transport real-time data streams such as video. Isochronous transfer is used to transport other non-time-critical data on a best-effort basis without latency or throughput guarantees.

At the lowest network level, IEEE 1394 defines three types of bus packets: a Physical Layer (PHY) packet, a Primary packet, and an Acknowledgement (Ack) packet. There are several different types of Primary packets. Primary packets contain a transaction code to distinguish them.

There are three different types of IEEE-1394 Data, Format 0 packets. The Chapter 10 standard refers to these as Type 0, Type 1, and Type 2. Type 0 packets are used to record bus events such as Bus Reset. Type 1 packets are used to record synchronous streams only (Primary packets with a TCODE of 0x0A). Type 2 packets are more general-purpose and are used to record all 1394 packets including PHY, Primary, and Ack packets.

The layout of the CSDW is shown in Figure 5-59. The **uPacketType** value indicates the packet body type. The **uSyncCode** value is the value of the 1394 synchronization code between transaction packets. The IEEE 1394 standard describes the synchronization code as "an application-specific control field" and "a synchronization point may be the defined as boundary

between video or audio frames, or any other point in the data stream the application may consider appropriate." The **uTransCnt** value is the number of separate transaction messages in the data packet.

```
struct Su1394F0_ChanSpec
    {
    uint32_t    uTransCnt    : 16;      // Transaction count
    uint32_t    Reserved     :  9;
    uint32_t    uSyncCode    :  4;      // Synchronization code
    uint32_t    uPacketType  :  3;      // Packet body type
    };
```

Figure 5-59. Type 0x58 IEEE-1394 Data, Format 0 CSDW

Type 0 and Type 1 data packets will not have an IPH. Type 2 data packets will have IPHs between transaction data messages with the data packet. The Type 2 IPH is an eight-byte time value. Time is represented in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

The IEEE 1394 standards contain quite a bit of example code and data structures in the C language to aid in data interpretation.

5.5.41 Type 0x59, IEEE-1394 Data, Format 1 (IEEE-1394 Physical Layer)

Format 1 IEEE 1394 data packets are used to record IEEE 1394 at the physical layer. All bus data and bus events can be recorded in Format 1 packets. The IEEE-1394 Data, Format 0, Type 2 data packet provides a similar capability for capturing all bus traffic at the physical level; however, this Format 1 packet provides more status information and is preferred over the Format 0, Type 2 data packet.

The layout of the CSDW is shown in Figure 5-60. The **uPacketCnt** value indicates the number of separate 1394 data messages in the data packet.

```
struct Su1394F1_ChanSpec
    {
    uint32_t    uPacketCnt   : 16;      // Number of messages
    uint32_t    Reserved     : 16;
    };
```

Figure 5-60. Type 0x58 IEEE-1394 Data, Format 1 CSDW

Individual 1394 data messages follow the CSDW. The format of the IPH is shown in Figure 5-61. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.. The **uDataLength** field is the length of the 1394 data message in bytes. The **bLBO** flag indicates that some 1394 packets were lost by the bus monitor due to an overflow in the local message buffer. The **uTrfOvf** value indicates a 1394 transfer protocol error. The **uStatus** field indicates the value of the eight-bit bus status transfer from the PHY to the link layer of the 1394 bus. Bus status transfers are used to signal a) bus reset indication; b) arbitration reset gap indication (both even and odd); c) subaction gap indication;

and d) cycle start indication (both even and odd). See IEEE 1394b-2002[25] Section 17.8 for more details about interpreting **`uStatus`**.

```
struct Su1394F1_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint32_t    uDataLength  : 16;      // Data length
    uint32_t    Reserved     :  1;      //
    uint32_t    bLBO         :  1;      // Local buffer overflow
    uint32_t    uTrfOvf      :  2;      // Transfer overflow
    uint32_t    uSpeed       :  4;      // Transmission speed
    uint32_t    uStatus      :  8;      // Status byte
    };
```

Figure 5-61. Type 0x59 IEEE-1394 Data, Format 1 Intra-Packet Header

The complete 1394 bus data message follows the IPH. The length of the 1394 data message is indicated in the IPH. If the data length is not a multiple of four, the data buffer will contain padding bytes to align the buffer on a four-byte boundary. The IEEE 1394 standards contain quite a bit of example code and data structures in the C language to aid in data interpretation.

5.5.42 Type 0x5A - 0x5F, IEEE-1394 Data, Format 2 - Format 7

Reserved for future use

5.5.43 Type 0x60, Parallel Data, Format 0

Parallel Data, Format 0 packets are used to record data bits received from a discrete parallel data interface. One data word can range from 2 to 128 bits in length. A parallel data packet can contain multiple parallel data words. Parallel Data, Format 0 packets support general-purpose parallel data and parallel data from the popular Ampex Digital Cartridge Recording System (DCRsi), which is a recording method and digital data interface developed by Ampex Data Systems. The DCRsi tape recording method is a transverse scan method with the tape heads embedded in the outer edge of a spinning disk placed perpendicular to the path of the tape. Data, as recorded on the DCRsi cartridge, is organized into discrete blocks, each assigned a unique address number and time stamped as it arrives at the recorder interface. The addressable block size is 4356 bytes. The electrical interface is byte-wide differential emitter-coupled logic. A simplified depiction of the interface is shown in Figure 5-62.

[25] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus: Amendment 2*. IEEE 1394b-2002. New York: Institute of Electrical and Electronics Engineers, 2002.

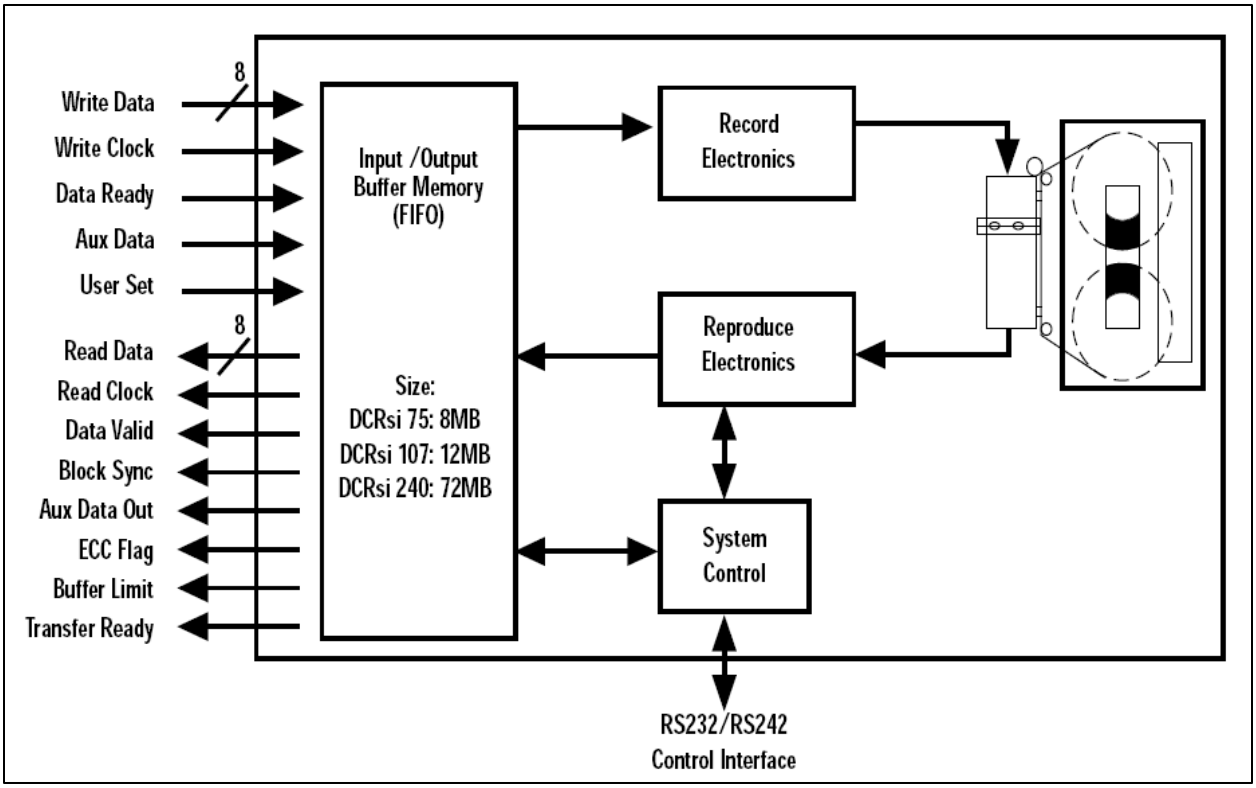

Figure 5-62. Ampex DCRsi Interface

The layout of the CSDW is shown in Figure 5-63. The **uType** field indicates the type or size of parallel data stored. For values between 2 and 128, **uType** indicates the number of bits per parallel data word. The value 254 indicates that the parallel data is in DCRsi format. Other values are reserved. When the data type is not DCRsi the **uScanNum** field is reserved and will have a value of 0x00. When the data type is DCRsi the **uScanNum** field contains the scan number value of the first scan stored in the packet for DCRsi data.

```
struct SuParallelF0_ChanSpec
    {
    uint32_t    uScanNum        : 24;       // Scan number
    uint32_t    uType           :  8;       // Data type
    };
```

Figure 5-63. Type 0x60 Parallel Data, Format 0 CSDW

Recorded parallel data follows the CSDW. There is no IPH. For general-purpose packets, bit padding is used to align recorded data on byte or word boundaries. There is no count for the number of parallel data words following the CSDW. This must be calculated from the data length in the header and the number of bytes per data word. See the Chapter 10 standard for complete details.

5.5.44 Type 0x61 - 0x67, Parallel Data, Format 1 - Format 7

Reserved for future use

5.5.45 Type 0x68, Ethernet Data, Format 0

Format 0 Ethernet data packets are used to record data frames from an Ethernet network. In general, an Ethernet data packet will contain multiple captured Ethernet data frames.

The layout of the CSDW is shown in Figure 5-64. The **uNumFrames** field indicates the number of Ethernet frames included in this data packet. The **uFormat** field indicates the format of the captured Ethernet frames. Format type **0x00** is described in Chapter 10 as "Ethernet Physical Layer" but in practice is better described as "Ethernet Data Link Layer" because support for physical layer features such as frame sync preamble and collision events are not described in the Chapter 10 standard. The IRIG 106-15 standard introduced the **uTTB** field to describe time tag reference bits in the Ethernet message.

```
struct SuEthernetF0_ChanSpec
    {
    uint32_t    uNumFrames      : 16;      // Number of frames
    uint32_t    Reserved1       :  9;
    uint32_t    uTTB            :  3;      // Time tag bits
    uint32_t    uFormat         :  4;      // Format of frames
    };
```

Figure 5-64. Type 0x68 Ethernet Data, Format 0 CSDW

Individual Ethernet data messages follow the CSDW. The format of the IPH is shown in Figure 5-65. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

```
struct SuEthernetF0_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uMsgDataLen     : 14;       // Message data length
    uint32_t    bLengthError    :  1;       // Data length error
    uint32_t    bDataCrcError   :  1;       // Data CRC error
    uint32_t    uNetID          :  8;       // Network identifier
    uint32_t    uSpeed          :  4;       // Ethernet speed
    uint32_t    uContent        :  2;       // Captured data content
    uint32_t    bFrameError     :  1;       // Frame error
    uint32_t    bFrameCrcError  :  1;       // Frame CRC error
    };
```

Figure 5-65. Type 0x68 Ethernet Data, Format 0 Intra-Packet Header

The **uDataLen** field is the length of the Ethernet data in bytes. The **uNetID** field is used to uniquely identify the physical network attachment point. The IRIG 106-13 standard introduced the **bLengthError**, **bCRCError**, and **bFrameCRCError** fields.

The **uSpeed** field indicates the bit rate at which the frame was captured by the recorder. Early coaxial cable-based Ethernet networks (10BASE5 and 10BASE2) required all network participants to operate at the same bit rate since they were sharing the same physical channel. Most modern Ethernet network topologies (10BASE-T, 100BASE-TX, etc.) are a star configuration with a single point-to-point link between the networked device and a network hub. In this case the network bit rate is the bit rate negotiated between the device (e.g., the recorder)

and the network hub. In this star topology different devices can operate at different bit rates on the same Ethernet network. The **uSpeed** field only indicates the bit rate of the link the data recorder has with the network hub. It does not imply the speed of any other devices on the network.

The **uContent** field indicates the type of data payload. The media access control (MAC) content type (0x00) indicates Ethernet data link layer frames, including the destination address, source address, type (in the case of Ethernet II) or length (in the case of IEEE 802.3), data, and frame check sequence. The data link frame preamble is not included, nor are other features of the physical layer, such as collisions or auto-negotiations. Content type (0x01) indicates that only the data payload portion of the Ethernet frame is stored. The **bFrameError** flag indicates that an unspecified error has occurred in the reception of the Ethernet frame.

5.5.46 Type 0x69, Ethernet Data, Format 1

Ethernet Data, Format 1 packets are used to hold UDP packets only. Although this can be used to hold UDP packets from any source, the common use for this data type is to hold ARINC-664 Part 7 data.

Since ARINC-664 is a redundant bus architecture, data is sent across multiple Ethernet networks simultaneously. The data receiver as well as the recorder choose which Ethernet interface to accept data from. Because of this recording the source and destination MAC addresses make little sense and so are not included in the recorded data.

The layout of the CSDW is shown in Figure 5-66. The **uNumFrames** field indicates the number of Ethernet frames included in this data packet. The **uIPHLength** field indicates the length of the IPH in bytes. The currently defined IPH is always 28 bytes.

```
struct SuEthernetF1_ChanSpec
    {
    uint32_t    uNumFrames      : 16;       // Number of frames
    uint32_t    uIPHLength      : 16;       // Intra-packet header length
    };
```

Figure 5-66. Type 0x69 Ethernet Data, Format 1 CSDW

Individual Ethernet data messages follow the CSDW. The format of the IPH is shown in Figure 5-67. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

```
struct SuEthernetF1_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uFlagBits       :  8;       // Flag bits
    uint32_t    uErrorBits      :  8;       // Error bits
    uint32_t    uMsgDataLen     : 16;       // Message data length
    uint32_t    uVirtualLinkID  : 16;       // Virtual link
    uint32_t    Reserved1       : 16;       //
    uint8_t     auSrcIP[4];                 // Source IP address
    uint8_t     auDstIP[4];                 // Source IP address
    uint32_t    uDstPort        : 16;       // Destination Port
    uint32_t    uSrcPort        : 16;       // Source Port
    };
```

Figure 5-67. Type 0x69 Ethernet Data, Format 1 Intra-Packet Header

Ethernet Data, Format 1 packets only contain the UDP payload portion of the Ethernet packet. Therefore IP parameters, such as source and destination IP addresses and source and destination port numbers, are stored in the IPH. The ARINC-664 standard defines a 16-bit Virtual Link value encoded into the Ethernet destination MAC address. This value is also stored in the IPH.

5.5.47 Type 0x6A - 0x6F, Ethernet Data, Format 2 - Format 7

Reserved for future use

5.5.48 Type 0x70, TSPI/CTS Data, Format 0

Time-space-position information (TSPI) and Combat Training System (CTS) position information typically use the Global Positioning System (GPS) as their sources. Any GPS data as defined by the National Marine Electronics Association (NMEA) and Radio Technical Commission for Maritime Services (RTCM) standards will be encapsulated in the Format 0 packet. The NMEA and RTCM standards specify the electrical signal requirements, data transmission protocol, and message/sentence formats for GPS data.

The layout of the CSDW is shown in Figure 5-68. The **`uIPTS`** flag indicates whether the IPTS is included. The **`uType`** field indicates the type of the NMEA-RTCM data:

0000 = NMEA 0183[26]
0001 = NMEA 0183-HS[27]
0010 = NMEA 2000[28]
0011 = RTCM SC104[29]

[26] National Marine Electronics Association. "NMEA 0183 Interface Standard." V 4.11. November 2018. May be superseded by update. Retrieved 13 August 2019. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.
[27] National Marine Electronics Association. "NMEA 0183-HS Interface Standard." V 1.01. n.d. May be superseded by update. Retrieved 10 August 2016. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.
[28] National Marine Electronics Association. "Standard for Serial-Data Networking of Marine Electronic Devices." Edition 3.101. March 2016. May be superseded by update. Retrieved 13 August 2019. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_2000.
[29] Radio Technical Commission for Maritime Services. "Standard for Networked Transport of RTCM via Internet Protocol (Ntrip)." RTCM 10410.1. Version 2.0 Amendment 1. June 2011. May be superseded by update. Retrieved

0010 - 1111 = Reserved

```
struct SuTSPICTSF0_ChanSpec
    {
    uint32_t    Reserved     : 27;
    uint32_t    uType        :  4;    // Type of data
    uint32_t    bIPTS        :  1;    // Intra-Packet Time Stamp enabled
    };
```

Figure 5-68. Type 0x70 TSPI/CTS Data, Format 0 CSDW

Individual data messages follow the CSDW. The format of the IPH is shown in Figure 5-69. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uLength** field specifies the length of the message in bytes.

```
struct SuTSPICTSF0_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uLength           : 16;     // Message length
    uint32_t    Reserved          : 16;
    };
```

Figure 5-69. Type 0x70 TSPI/CTS Data, Format 0 Intra-Packet Header

5.5.49 Type 0x71, TSPI/CTS Data, Format 1

Air Combat Maneuvering Instrumentation (ACMI) data as defined by the European Air Group (EAG) interface control document (ICD) DF29125[30] for post-mission interoperability will be encapsulated in the Format 1 packet. The EAG ACMI ICD defines the data contents and organization. Electrical signal requirements and data transmission protocol are not defined in DF29125 or in the Chapter 10 format.

The layout of the CSDW is shown in Figure 5-70. The **bIPTS** flag indicates whether the IPTS is included. The **uContent** field indicates the type of the EAG ACMI data:

00 = TSPI data only (no static data or pod ID)
01 = Contains pod ID and static data

```
struct SuTSPICTSF1_ChanSpec
    {
    uint32_t    Reserved     : 29;
    uint32_t    uContent     :  2;    // Packet contents
    uint32_t    bIPTS        :  1;    // Intra-Packet Time Stamp enabled
    };
```

Figure 5-70. Type 0x71 TSPI/CTS Data, Format 1 CSDW

13 August 2019. Available for purchase at http://www.rtcm.org/differential-global-navigation-satellite--dgnss--standards.html.

[30] European Air Group. "European Air Group Interface Control Document for Post Mission Interoperability." DF29125 Draft A Issue 01. July 2004. Retrieved 13 August 2019. Available to RCC members with Private Portal access at https://wsdmext.wsmr.army.mil/site/rccpri/Limited_Distribution_References/DF29125.pdf.

Individual data messages follow the CSDW. The format of the IPH is shown in Figure 5-71. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uLength** field specifies the length of the message in bytes.

```
struct SuTSPICTSF1_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uLength           : 16;     // Message length
    uint32_t    Reserved          : 16;
    };
```

Figure 5-71. Type 0x71 TSPI/CTS Data, Format 1 Intra-Packet Header

5.5.50 Type 0x72, TSPI/CTS Data, Format 2

Air Combat Test and Training System (ACTTS) data as defined by the USAF ACTTS interface specification WMSP 98-01[31] will be encapsulated in the Format 2 packet. The ACTTS interface specification defines the unique signal interface requirements for the air-to-air, air-to-ground, ground-to-air data links, and aircraft information subsystem recording formats. The ACTTS WMSP 98-01 establishes the requirements for the information recorded on the different data transfer units used by the various ACTTS airborne subsystems to support both tethered and rangeless operations.

When encapsulating ACTTS message/word format, data messages or words will not span packets. Each new packet will start with a full message and not a partial message or word.

The layout of the CSDW is shown in Figure 5-72. The **bIPTS** flag indicates whether the IPTS is included. The **uFormat** field indicates the type of the ACTTS data:

0000 = Kadena Interim Training System (KITS) Recording Formats
0001 = Alpena KITS Recording Formats
0010 = USAF Europe Rangeless Interim Training System Recording Formats
0011 = Alaska Air Combat Training System (ACTS) Upgrade Recording Formats
0100 = Goldwater Range Mission and Debriefing System Recording Formats
0101 = P4RC Recording Formats
0110 = Nellis ACTS Range Security Initiative Recording Formats
0111 = P4RC+P5 CTS Participant Subsystem Recording Formats
1000 = P5 CTS Recording Formats
1001 - 1111 = Reserved

[31] Range Instrumentation System Program Office, Air Armament Center. "Interface Specification for the USAF Air Combat Test and Training System (ACTTS) Air-to-Ground, Air-to-Air, Ground-to-Air Data Links, and AIS Recording Formats." WMSP 98-01, Rev A, Chg 1. 19 May 2003. Retrieved 3 June 2015. Available to RCC members with Private Portal access at https://wsdmext.wsmr.army.mil/site/rccpri/Limited_Distribution_References/WMSP_98-01.doc.

```
struct SuTSPICTSF2_ChanSpec
    {
    uint32_t    Reserved     : 27;
    uint32_t    uFormat      :  4;    // Format of data
    uint32_t    bIPTS        :  1;    // Intra-Packet Time Stamp enabled
    };
```

Figure 5-72. Type 0x72 TSPI/CTS Data, Format 2 CSDW

Individual data messages follow the CSDW. The format of the IPH is shown in Figure 5-73. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8. The **uLength** field specifies the length of the message in bytes.

```
struct SuTSPICTSF2_Header
    {
    uint8_t     aubyIntPktTime[8];          // Reference time
    uint32_t    uLength           : 16;     // Message length
    uint32_t    Reserved          : 16;
    };
```

Figure 5-73. Type 0x72 TSPI/CTS Data, Format 2 Intra-Packet Header

5.5.51 Type 0x73-0x77, TSPI/CTS Data, Format 3 – Format 7

Reserved for future use

5.5.52 Type 0x78, Controller Area Network Bus

Controller Area Network Data, Format 0 packets are used to record data bits received from one or more CAN bus interfaces. The layout of the CSDW is shown in Figure 5-74. The **uMessages** field contains a binary value indicating the number of messages included in the packet.

```
struct SuCANBus_ChanSpec
    {
    uint32_t    uMessages       : 16;      // Message counter
    uint32_t    uReserved       : 16;
    };
```

Figure 5-74. Type 0x78 CAN Bus CSDW

After the CSDW, CAN bus data is inserted into the packet. Each CAN bus message is preceded by an IPH that has an IPDH and an IPTS. The IPDH is composed of an intra-packet message header and an intra-packet ID word. The length of the IPH is fixed at 16 bytes (128 bits) positioned contiguously, with the layout shown in Figure 5-75. The **uMsgLength** field contains a binary value representing the length of the number of the valid bytes in the rest of the message that follows the intra-packet message header. The message length will be 4-12 bytes (4 bytes for the intra-packet ID word + 0-8 bytes data content of the CAN bus message). The **uSubchannel** field contains a binary value that represents the subchannel number belonging to the message that follows the ID word when the channel ID in the packet header defines a group of subchannels. Zero means first and/or only subchannel, which is valid for the CAN bus. The **bFmtError** flag indicates a message protocol error condition. The **bDataError** flag indicates an error in the data such as a parity error. **uCANBUsID** is the standard 11-bit or

extended 29-bit CAN Bus ID address. The **bRTR** flag indicates a remote transfer request. The **bIDE** flag indicates whether to use the extended CAN identifier:

0 = 11-bit standard CAN identifier (CAN ID word bits 10-0)
1 = 29-bit extended CAN identifier (CAN ID word bits 28-0)

```
struct SuCANBus_Header
    {
    uint8_t     aubyIntPktTime[8];        // Reference time
    uint32_t    uMsgLength      :  4;     // Message length
    uint32_t    uReserved1      : 12;
    uint32_t    uSubChannel     :  8      // Subchannel number
    uint32_t    uReserved2      :  6;
    uint32_t    bFmtError       :  1;     // Format error flag
    uint32_t    bDataError      :  1;     // Data error flag
    uint32_t    uCANBusID       : 29;     // CAN Bus ID
    uint32_t    uReserved3      :  1;     //
    uint32_t    bRTR            :  1;     // Remote transfer request bit
    uint32_t    bIDE            :  1;     // Extended CAN identifier
    };
```

Figure 5-75. Type 0x78 CAN Bus Intra-Packet Header

5.5.53 Type 0x79, Fibre Channel Data, Format 0

Fibre Channel Data, Format 0 packets are used to store Fibre Channel data from the Fibre Channel Physical Layer. Fibre Channel, Format 0 data includes all physical layer bits, including Start-of-Frame delimiter, End-of-Frame delimiter, and CRC word of the frame.

Fibre Channel is logically a bi-directional, point-to-point, serial data channel, structured for high-performance capability. Fibre Channel may be implemented using any combination of the following three topologies:

a) A point-to-point link between two ports;

b) A set of ports interconnected by a switching network called a Fabric; and

c) A set of ports interconnected with a loop topology.

The Fibre Channel Data, Format 0 format packet supports both an "unstripped" and "stripped" format. With unstripped format all frame fields are included in the data portion of the packet. In stripped format Start-of-Frame delimiter, End-of-Frame delimiter, any optional header fields, and CRC words of the frame are removed. Only the data payload portion of the packet is stored. The Start-of-Frame delimiter and End-of-Frame delimiter can still be found in the IPH.

The layout of the CSDW is shown in Figure 5-76. The **uNumFrames** field contains a binary value indicating the number of frames included in the packet. The **uFormat** field contains the format of the data portion of the packet. Currently only the Fibre Channel Physical Layer format is supported.

```
struct SuFibreChanF0_ChanSpec
    {
    uint32_t    uNumFrames      : 16;     // Number of frames
    uint32_t    Reserved        : 12;
    uint32_t    uFormat         :  4;     // Frame data format
    };
```

Figure 5-76. Type 0x79 Fibre Channel, Format 0 CSDW

Individual Fibre Channel data frame messages follow the CSDW. The format of the IPH is shown in Figure 5-77. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

```
struct SuFibreChanF0_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint32_t    uFrameLen       : 12;   // Frame length in bytes
    uint32_t    uStartOfFrame   :  4;   // Start of frame delimiter
    uint32_t    uEndOfFrame     :  3;   // End of frame delimiter
    uint32_t    Reserved        :  5;
    uint32_t    uTopology       :  2;   // Fibre Channel topology
    uint32_t    uContent        :  2;   // Fibre Channel frame content
    uint32_t    bNonStrippedMode : 1;   // Stripped / Non-stripped mode flag
    uint32_t    bOverrunError   :  1;   // Overrun error flag
    uint32_t    bCrcError       :  1;   // CRC error flag
    uint32_t    bFramingError   :  1;   // Framing error flag
    };
```

Figure 5-77. Type 0x79 Fibre Channel, Format 0 Intra-Packet Header

The **uFrameLen** field is the length of the data portion of the packet in bytes. Note that in unstripped mode this value is the numer of bytes in the entire frame and should be a value divisible by four. In stripped mode this value is the number of bytes in the data payload portion of the frame. The **uStartOfFrame** and **uEndOfFrame** fields hold the values of the frame delimeter fields in the physical frame. The **uTopology** field represents the topology of the Fibre Channel. Currenly only passive topology is supported. The **uContent** field indicates whether the data is a data frame or a link control frame. The **bNonStrippedMode** flag indicates stripped or unstripped storage mode. The **bOverrunError**, **bCrcError**, and **bFramingError** flags indicate their respective data error conditions.

5.5.54 Type 0x7A, Fibre Channel Data, Format 1

Fibre Channel Data, Format 1 packets are used to store Fibre Channel data from the Fibre Channel Framing and Signalling layer. This transmission protocol is separate and distinct from the FC control protocol.

The Fibre Channel Framing and Signalling layer provides a general transport mechanism for upper-level protocols such as SCSI command sets, IP, and others.

The layout of the CSDW is shown in Figure 5-78. The **uNumFrames** field contains a binary value indicating the number of frames included in the packet.

```
struct SuFibreChanF1_ChanSpec
    {
    uint32_t    uNumFrames      : 16;       // Number of frames
    uint32_t    Reserved        : 16;
    };
```

Figure 5-78. Type 0x7A Fibre Channel, Format 1 CSDW

Individual Fibre Channel data frame messages follow the CSDW. The format of the IPH is shown in Figure 5-79. The **aubyIntPktTime** value is an eight-byte representation of time in 48-bit relative time format derived from the RTC shown in Figure 5-6, 64-bit extended relative time format shown in Figure 5-9, absolute time in Chapter 4 format shown in Figure 5-7, or IEEE-1588 time format shown in Figure 5-8.

```
struct SuFibreChanF1_Header
    {
    uint8_t     aubyIntPktTime[8];      // Reference time
    uint32_t    uFrameLen       : 12;   // Frame length in bytes
    uint32_t    uStartOfFrame   :  4;   // Start of frame delimiter
    uint32_t    uEndOfFrame     :  3;   // End of frame delimiter
    uint32_t    Reserved        : 10;
    uint32_t    bOverrunError   :  1;   // Overrun error flag
    uint32_t    bCrcError       :  1;   // CRC error flag
    uint32_t    bFramingError   :  1;   // Framing error flag
    };
```

Figure 5-79. Type 0x7A Fibre Channel, Format 1 Intra-Packet Header

The **uFrameLen** field is the length of the data portion of the packet in bytes. In stripped mode this value is the number of bytes in the data payload portion of the frame. The **uStartOfFrame** and **uEndOfFrame** fields hold the values of the frame delimeter fields in the physical frame. The **bOverrunError**, **bCrcError**, and **bFramingError** flags indicate their respective data error conditions.

5.5.55 Type 0x7B-0x7F, Fibre Channel Data, Formats 2-7

Reserved for future use

5.6 Time Interpretation

Chapter 10 defines a 48-bit RTC as the basis for all packet and message time stamps. The RTC clock is 10 MHz, resulting in a clock resolution of 100 nanoseconds. There is no constraint on the absolute value of the RTC and at recorder power on it could be initialized to zero or some random number. Some recorder vendors will preset the RTC to a value based on absolute clock time, but for interoperability reasons it is unwise to assume the RTC value will be related to absolute clock time in any meaningful fashion.

Absolute clock time comes into a recorder and is recorded much like any other data source. In fact, there may be multiple time sources recorded. Time Data, Format 1 data packets are used to record input time signals. Since Time Data, Format 1 packets contain both the absolute input time value and the RTC clock value at the instant the absolute time was valid, these packets can be used to relate RTC values to the input absolute time source.

For example, if a time packet is recorded with an RTC value of 1,000,000 and an absolute time value of 100:12:30:25.000, then the clock time of a subsequent data packet with an RTC value of 1,150,000 could be deduced to be 100:12:30:25.015 (150,000 clock tics x 100 nanoseconds per tic = 15 milliseconds).

When multiple time channels are available, it is incumbent on the programmer or data analyst to determine and select the best source of time for a particular data set.

The IRIG 106-15 standard introduced a 64-bit ERTC, which is driven from a 1-GHz clock, providing a time resolution of 1 nanosecond. Interpretation of the ERTC value is the same as with the 48-bit 10-MHz RTC. The ERTC is stored in the optional packet secondary header.

5.7 Index and Event Records

Often times it is useful to make an in-memory version of the data file index. This allows rapid access to recorded data packets based on time or the occurrence of events. Below is a general algorithm for reading all root and node index packets.

1. If "**R-x\IDX\E**" not equal to "T" then index does not exist.
2. Move read pointer to last packet of data file. Store file offset of this packet.
3. If last packet data type does not equal 0x03 (Computer-Generated Data, Format 3) then index does not exist.
4. Get the index count from the CSDW.
5. For each root index contained in the packet:
    a. Read the node index offset value.
    b. Move the read pointer to the node index offset value.
    c. Read the node index packet.
    d. Get the node index count from the CSDW.
    e. For each node index contained in the packet read and store the time stamp, channel ID, data type, and data packet offset values.
6. Read last root node index. If offset value is equal to current root node packet offset (stored in Step 2) then done.
7. Else move read pointer to the next root index packet offset value.
8. Read the next root index packet.
9. Go to Step 4.

5.8 Data Streaming

Chapter 10 recorders can stream their data over one of their download interface network ports using UDP/IP and Chapter 10 UDP transfer headers. This is normally done over an Ethernet port, but any network connection that supports UDP/IP can use this method.

The **.PUBLISH** command is used to control data streaming. Chapter 6 defines the use of **.PUBLISH** and has numerous examples of its use.

Data can be streamed to one or more specific unicast and multicast IP addresses or a broadcast address. Different channels can be addressed to different addresses.

It's common to publish different groups of data to different multicast groups. According to RFC 3171,[32] addresses 224.0.0.0 to 239.255.255.255 are designated as multicast addresses. Different multicast address regions are designated for different purposes. According to RFC

[32] Internet Engineering Task Force. "IANA Guidelines for IPv4 Multicast Address Assignments." RFC 3171. August 2001. Obsoleted by RFC 5771. Retrieved 13 August 2019. Available at https://datatracker.ietf.org/doc/rfc3171/.

2365,[33] Chapter 10 data streaming should be directed to multicast addresses in the local scope address range 239.255.0.0 to 239.255.255.255.

All IP multicast packets are delivered by using the Ethernet MAC address range 01:00:5e:00:00:00 - 01:00:5e:7f:ff:ff. This is 23 bits of available address space. The lower 23 bits of the 28-bit multicast IP address are mapped into the 23 bits of available Ethernet address space. This means that there is ambiguity in delivering packets. If two hosts on the same subnet each subscribe to a different multicast group whose address differs only in the first 5 bits, Ethernet packets for both multicast groups will be delivered to both hosts, requiring the network software in the hosts to discard the unrequired packets. If multiple multicast addresses are used, be careful to choose multicast addresses that will result in different Ethernet multicast addresses.

Multicast data is filtered by the Ethernet controller hardware, only passing subscribed packets to the software driver for decoding. This improves performance under high network traffic loads. Ethernet controllers only have a limited number of multicast addresses they can filter. A common hardware limitation is 16 multicast addresses. If a workstation needs to subscribe to more multicast addresses than the Ethernet hardware can support, then all multicast traffic is passed to the software driver for filtering, negating the benefit of multicast filtering in hardware.

The size of a UDP packet is represented by a 16-bit value in the IPv4 IP and UDP headers, but some software implementations treat this as a signed value with a maximum value of 2^15 or 32,768. Because of this, the maximum size of a Chapter 10 streaming packet should be no more than 32,724 bytes. Physical networks have a maximum transfer unit (MTU), which is the largest data packet they can carry. If a UDP packet has a size larger than the network MTU, it will be fragmented into smaller packets by the IP software driver before sending them over the underlying physical network. The fragmented UDP packets are then reassembled into a larger packet by the IP software driver at the receiving end. There is a performance penalty for this fragmentation and reassembly. Better performance may be achieved by choosing a UDP packet small enough to avoid fragmentation and reassembly. Regular Ethernet supports a maximum size of 1500 bytes of data payload (IP header, UDP header, and UDP data) but some newer Ethernet technologies support larger jumbo frames.

Chapter 10 data packets are sent in a UDP/IP packet by prepending a UDP transfer header to the UDP data payload. Chapter 10 data packet(s) smaller than the 32-kb maximum size will prepend the non-segmented UDP transfer header shown in Figure 5-80. A Chapter 10 data packet larger than the 32-kb maximum size will need to be segmented before transmission, and will prepend the segmented UDP transfer header shown in Figure 5-81. The IPv6 format supports large data packets, negating the need for segmented data packets.

```
struct SuUDP_Transfer_Header_F1_NonSeg
    {
    uint32_t    uFormat         : 4;    // Streaming version
    uint32_t    uMsgType        : 4;    // Type of message
    uint32_t    uSeqNum         :24;    // UDP sequence number
    };
```

Figure 5-80. UDP Transfer Header Format 1, Non-Segmented Data

[33] Internet Engineering Task Force. "Administratively Scoped IP Multicast." RFC 2365. July 1998. May be superseded by update. Retrieved 13 August 2019. Available at https://datatracker.ietf.org/doc/rfc2365/.

```
struct SuUDP_Transfer_Header_F1_Seg
    {
    uint32_t    uFormat          : 4;    // Streaming version
    uint32_t    uMsgType         : 4;    // Type of message
    uint32_t    uUdpSeqNum       :24;    // UDP sequence number
    uint32_t    uChID            :16;    // Channel ID
    uint32_t    uChanSeqNum      : 8;    // Channel sequence number
    uint32_t    uReserved        : 8;
    uint32_t    uSegmentOffset;          // Segment offset
    };
```

Figure 5-81. UDP Transfer Header Format 1, Segmented Data

Computer-Generated Data, Format 3 (recording index) packets are meaningless in a network data stream. It is necessary that they be transmitted so that Channel ID 0 data packets will have contiguous sequence numbers for error detection. They should be ignored, though, when received.

The IRIG 106-17 standard introduced two new streaming formats: Format 2 and Format 3. It also introduced Computer Generated Data Packet, Format 4 (described in Section 5.5.5), which is now the preferred packet type for transmitting TMATS setup records.

Format 2 has been defined to document existing legacy streaming implementations but is not recommended for future implementations. The layout of the Format 2 UDP transfer header is shown in Figure 5-82. Note that data fields in the Format 2 structure are big-endian.

```
struct SuUDP_Transfer_Header_F2
    {
    uint32_t    uFormat          :  4;    // Streaming version
    uint32_t    uMsgType         :  4;    // Type of message
    uint32_t    uUdpSeqNum       : 24;    // UDP sequence number
    uint32_t    uPacketSize      : 24;    // Packet size in bytes
    uint32_t    uSegmentOffset1  :  8;
    uint32_t    uChanID;         : 16;
    uint32_t    uSegmentOffset2  : 16;
    };
```

Figure 5-82. UDP Transfer Header Format 2

Format 3 is designed to support distributed data acquisition systems. It supports stream to and from a recorder. It is the recommended streaming format for all new designs. The layout of the Format 3 UDP transfer header is shown in Figure 5-83.

```
struct SuUDP_Transfer_Header_F3
    {
    uint32_t    uFormat          :  4;    // Streaming version
    uint32_t    uSrcIdLen        :  4;    // Type of message
    uint32_t    uReserved        :  8;    // UDP sequence number
    uint32_t    uPktStartOffset  : 16;    // Start of Ch 11 packet
    uint32_t    uUdpSeqNum       : 24;    // UDP sequence number
    uint32_t    uSrcId           :  8;    // Source identifier
    };
```

Figure 5-83. UDP Transfer Header Format 3

This page intentionally left blank.

CHAPTER 6

Conformance to IRIG 106

The Chapter 10 standard sets forth requirements for digital recorders. The IRIG 106 Chapter 10.3.1 summarizes requirements for a recorder to be considered 100 percent compliant with the standard. There is also a number of features of Chapter 10 that are considered optional, which are not required to be implemented, but if they are implemented must be IAW the Chapter 10 standard.

Rather than reiterate all the requirements of Chapter 10, Table 6-1 and Table 6-2 present a brief outline of the major requirement areas and lists those portions of Chapter 10 that have been identified as optional.

Table 6-1. Physical Interface Requirements

Ch 10 Section	Required	Optional	
10.3, 10.4.1	1		Recorder Fibre Channel
10.4.2	✓		Fibre Channel SCSI
10.4.1	✓		Data Download
10.3.11		✓	Data streaming
10.3.8, 10.7		✓	Configuration with TMATS
10.7.1		✓	Recorder Control and Status
10.4.2	1		Recorder IEEE-1394B
10.4.2.2	✓		SCSI/SBP-2
10.4, 10.9	✓		Data Download
10.3.11		✓	Data streaming
10.3.8, 10.7		✓	Configuration with TMATS
10.7.1		✓	Recorder control and status
10.4, 10.4.3	1		Recorder Ethernet (on board)
10.4	✓		Data Download
10.3.11		✓	Data streaming
10.3.8, 10.7		✓	Configuration with TMATS
10.7.1		✓	Recorder Control And Status
10.3.5, 10.3.6		✓	RMM
10.9.5			IEEE-1394B Bilingual Socket
10.9	✓		IEEE-1394B SCSI/SBP-2
10.4, 10.9	✓		Data Download
10.3.8, 10.7		✓	Configuration with TMATS
10.3.2, 10.7.10		✓	Discrete Lines
10.7.10	✓		Recorder control and status
10.3.2	✓		RS-232 and 422 Full Duplex Communication
10.7	✓		Configuration with TMATS
10.7	✓		Recorder Control And Status
10.3	✓		External Power Port

1 - Either Fibre Channel or IEEE 1394B is required for on-board recorders; other network interfaces are optional. Ethernet is required for ground-based recorders; other network interfaces are optional.

Table 6-2. Logical Interface Requirements

Logical Interfaces			
Ch 10 Section	**Required**	**Optional**	
10.5	✓		Media File Interface
10.5	✓		Directory & File Table Entries
10.3.7		✓	STANAG fields - File Size, File Create Date, File Close Time
10.6	✓		Packetization & Data Format
10.6.1		✓	Secondary header
10.6.1		✓	Filler for packet length
10.6.1		✓	Data checksum
10.6.1		✓	Intra-packet headers and time stamps

Appendix A

Summary of Ch 9 TMATS Code Values Supported

Legend	
(blank)	Not used
X	Used
R	Required
O	Only required if data type used
P	Only required Data Packing Option (R-x\PDP-n) is Packed or Unpacked

Note: In the table below "R", "O", and "P" have only been entered for 2013, 2015, 2017, and 2019.

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
PROGRAM NAME	G\PN	X	X	X	X	X	X	X	X	X
TEST ITEM	G\TA	X	X	X	X	X	X	X	X	X
TMATS FILE NAME	G\FN				X	X	X	X	X	X
IRIG 106 REVISION LEVEL	G\106	X	X	X	X	X	R	R	R	R
ORIGINATION DATE	G\OD	X	X	X	X	X	X	X	X	X
REVISION NUMBER	G\RN	X	X	X	X	X	X	X	X	X
REVISION DATE	G\RD	X	X	X	X	X	X	X	X	X
UPDATE NUMBER	G\UN	X	X	X	X	X	X	X	X	X
UPDATE DATE	G\UD	X	X	X	X	X	X	X	X	X
TEST NUMBER	G\TN	X	X	X	X	X	X	X	X	X
NUMBER OF POINTS OF CONTACT	G\POC\N	X	X	X	X	X	X	X	X	X
NAME	G\POC1-n	X	X	X	X	X	X	X	X	X
AGENCY	G\POC2-n	X	X	X	X	X	X	X	X	X
ADDRESS	G\POC3-n	X	X	X	X	X	X	X	X	X
TELEPHONE	G\POC4-n	X	X	X	X	X	X	X	X	X
NUMBER OF DATA SOURCES	G\DSI\N	X	X	X	X	X	R	R	R	R
DATA SOURCE ID	G\DSI-n	X	X	X	X	X	R	R	R	R
DATA SOURCE TYPE	G\DST-n	X	X	X	X	X	R	R	R	R
DATA SOURCE SECURITY CLASSIFICATION	G\DSC-n						X	X	X	X
TEST DURATION	G\TI1	X	X	X	X	X	X	X	X	X
PRE-TEST REQUIREMENT	G\TI2	X	X	X	X	X	X	X	X	X
POST-TEST REQUIREMENT	G\TI3	X	X	X	X	X	X	X	X	X
SECURITY CLASSIFICATION	G\SC	X	X	X	X	X	X	X	X	X
MESSAGE DIGEST/CHECKSUM	G\SHA							X	X	X
COMMENTS	G\COM	X	X	X	X	X	X	X	X	X
DATA SOURCE ID	T-x\ID	X	X	X	X	X	X	O	O	O
TRANSMITTER ID	T-x\TID	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
FREQUENCY	T-x\RF1	X	X	X	X	X	X	X	X	X
RF BANDWIDTH	T-x\RF2	X	X	X	X	X	X	X	X	X
DATA BANDWIDTH	T-x\RF3	X	X	X	X	X	X	X	X	X
MODULATION TYPE	T-x\RF4	X	X	X	X	X	X	X	X	X
TOTAL CARRIER MODULATION	T-x\RF5	X	X	X	X	X	X	X	X	X
POWER (RADIATED)	T-x\RF6	X	X	X	X	X	X	X	X	X
NUMBER OF SUBCARRIERS	T-x\SCO\N	X	X	X	X	X	X	X	X	X
SUBCARRIER NUMBER	T-x\SCO1-n	X	X	X	X	X	X	O	O	O
MODULATION INDEX	T-x\SCO2-n	X	X	X	X	X	X	X	X	X
MODULATOR NON-LINEARITY	T-x\RF7	X	X	X	X	X	X	X	X	X
BANDWIDTH	T-x\PMF1	X	X	X	X	X	X	X	X	X
SLOPE	T-x\PMF2	X	X	X	X	X	X	X	X	X
TYPE	T-x\PMF3	X	X	X	X	X	X	X	X	X
TRANSMIT ANTENNA TYPE	T-x\AN1	X	X	X	X	X	X	X	X	X
TRANSMIT POLARIZATION	T-x\AN2	X	X	X	X	X	X	X	X	X
ANTENNA LOCATION	T-x\AN3	X	X	X	X	X	X	X	X	X
DOCUMENT	T-x\AP	X	X	X	X	X	X	X	X	X
NAME	T-x\AP\POC1	X	X	X	X	X	X	X	X	X
AGENCY	T-x\AP\POC2	X	X	X	X	X	X	X	X	X
ADDRESS	T-x\AP\POC3	X	X	X	X	X	X	X	X	X
TELEPHONE	T-x\AP\POC4	X	X	X	X	X	X	X	X	X
IF BANDWIDTH	T-x\GST1	X	X	X	X	X	X	X	X	X
BASEBAND COMPOSITE BANDWIDTH	T-x\GST2	X	X	X	X	X	X	X	X	X
AGC TIME CONSTANT	T-x\GST3	X	X	X	X	X	X	X	X	X
MGC GAIN SET POINT	T-x\GST4	X	X	X	X	X	X	X	X	X
AFC/APC	T-x\GST5	X	X	X	X	X	X	X	X	X
TRACKING BANDWIDTH	T-x\GST6	X	X	X	X	X	X	X	X	X
POLARIZATION RECEPTION	T-x\GST7	X	X	X	X	X	X	X	X	X
DISCRIMINATOR BANDWIDTH	T-x\FM1	X	X	X	X	X	X	X	X	X
DISCRIMINATOR LINEARITY	T-x\FM2	X	X	X	X	X	X	X	X	X
PHASE LOCK LOOP BANDWIDTH	T-x\PLL	X	X	X	X	X	X	X	X	X
COMMENTS	T-x\COM	X	X	X	X	X	X	X	X	X
SOURCE ID	R-x\ID	X	X	X	X	X	R	R	R	R
TAPE/STORAGE ID	R-x\RID	X	X	X	X	X	R	R	R	R
TAPE/STORAGE DESCRIPTION	R-x\R1	X	X	X	X	X	X	X	X	X
TAPE/STORAGE TYPE	R-x\TC1	X	X	X	X	X	X	X	X	X
TAPE/STORAGE MANUFACTURER	R-x\TC2	X	X	X	X	X	X	X	X	X
TAPE/STORAGE CODE	R-x\TC3	X	X	X	X	X	X	X	X	X
RECORDER-REPRODUCER MEDIA LOCATION	R-x\RML				X	X	R	R	R	R
EXTERNAL RMM BUS SPEED	R-x\ERBS				X	X	R	O	O	O

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
TAPE WIDTH	R-x\TC4	X	X	X	X	X	X	X	X	X
TAPE HOUSING	R-x\TC5	X	X	X	X	X	X	X	X	X
TYPE OF TRACKS	R-x\TT	X	X	X	X	X	X	X	X	X
NUMBER OF TRACKS/CHANNELS	R-x\N	X	X	X	X	X	R	R	R	R
RECORD SPEED	R-x\TC6	X	X	X	X	X	X	X	X	X
DATA PACKING DENSITY	R-x\TC7	X	X	X	X	X	X	X	X	X
TAPE REWOUND	R-x\TC8	X	X	X	X	X	X	X	X	X
NUMBER OF SOURCE BITS	R-x\NSB		X	X	X	X	R	R	R	R
TAPE DRIVE/STORAGE MANUFACTURER	R-x\RI1	X	X	X	X	X	X	X	X	X
TAPE DRIVE/STORAGE MODEL	R-x\RI2	X	X	X	X	X	X	X	X	X
ORIGINAL TAPE/STORAGE	R-x\RI3	X	X	X	X	X	R	R	R	R
DATE AND TIME CREATED	R-x\RI4	X	X	X	X	X	X	X	X	X
NAME	R-x\POC1		X	X	X	X	X	X	X	X
AGENCY	R-x\POC2		X	X	X	X	X	X	X	X
ADDRESS	R-x\POC3		X	X	X	X	X	X	X	X
TELEPHONE	R-x\POC4		X	X	X	X	X	X	X	X
DATE OF DUB	R-x\RI5	X	X	X	X	X	O	O	O	O
NAME	R-x\DPOC1	X	X	X	X	X	X	X	X	X
AGENCY	R-x\DPOC2	X	X	X	X	X	X	X	X	X
ADDRESS	R-x\DPOC3	X	X	X	X	X	X	X	X	X
TELEPHONE	R-x\DPOC4	X	X	X	X	X	X	X	X	X
POST PROCESS MODIFIED RECORDING	R-x\RI6				X	X	R	R	R	R
POST PROCESS MODIFICATION TYPE	R-x\RI7				X	X	O	O	O	O
DATE AND TIME OF MODIFICATION	R-x\RI8				X	X	O	O	O	O
NAME	R-x\MPOC1				X	X	X	X	X	X
AGENCY	R-x\MPOC2				X	X	X	X	X	X
ADDRESS	R-x\MPOC3				X	X	X	X	X	X
TELEPHONE	R-x\MPOC4				X	X	X	X	X	X
CONTINUOUS RECORDING ENABLED	R-x\CRE				X	X	R	R	R	R
RECORDER-REPRODUCER SETUP SOURCE	R-x\RSS				X	X	R	R	R	R
RECORDER SERIAL NUMBER	R-x\RI9				X	X	X	X	X	X
RECORDER FIRMWARE REVISION	R-x\RI10				X	X	X	X	X	X
NUMBER OF MODULES	R-x\RIM\N				X	X	X	X	X	X
MODULE ID	R-x\RIMI-n				X	X	X	X	X	X
MODULE SERIAL NUMBER	R-x\RIMS-n				X	X	X	X	X	X
MODULE FIRMWARE REVISION	R-x\RIMF-n				X	X	X	X	X	X
NUMBER OF RMM?S	R-x\RMM\N				X	X	X	X	X	X
RMM IDENTIFIER	R-x\RMMID-n				X	X	X	X	X	X
RMM SERIAL NUMBER	R-x\RMMS-n				X	X	X	X	X	X
RMM FIRMWARE REVISION	R-x\RMMF-n				X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
NUMBER OF ETHERNET INTERFACES	R-x\EI\N						O	O	O	O
ETHERNET INTERFACE NAME	R-x\EINM-n						O	O	O	O
PHYSICAL ETHERNET INTERFACE	R-x\PEIN-n								O	O
ETHERNET INTERFACE LINK SPEED	R-x\EILS-n								O	O
ETHERNET INTERFACE TYPE	R-x\EIT-n						O	O	O	O
ETHERNET INTERFACE IP ADDRESS	R-x\EIIP-n						O	O	O	O
NUMBER OF ETHERNET INTERFACE PORTS	R-x\EIIP\N-n						O	O	O	O
PORT ADDRESS	R-x\EI\PA-n-m						O	O	O	O
PORT TYPE	R-x\EI\PT-n-m						O	O	O	O
NUMBER OF CHANNEL GROUPS	R-x\CG\N						O	O	O	O
CHANNEL GROUP NAME	R-x\CGNM-n						O	O	O	O
CHANNEL GROUP STREAM NUMBER	R-x\CGSN-n						O	O	O	O
NUMBER OF GROUP CHANNELS	R-x\CGCH\N-n						O	O	O	O
GROUP CHANNEL NUMBER	R-x\CGCN-n-m						O	O	O	O
NUMBER OF DRIVES	R-x\DR\N						O	O	O	O
DRIVE NAME	R-x\DRNM-n						O	O	O	O
DRIVE NUMBER	R-x\DRN-n						O	O	O	O
DRIVE BLOCK SIZE	R-x\DRBS-n						O	O	O	O
NUMBER OF DRIVE VOLUMES	R-x\DRVL\N-n						O	O	O	O
VOLUME NAME	R-x\VLNM-n-m						O	O	O	O
VOLUME NUMBER	R-x\VLN-n-m						O	O	O	O
VOLUME BLOCKS TO ALLOCATE	R-x\VLBA-n-m						O	O	O	O
VOLUME NUMBER OF BLOCKS	R-x\VLNB-n-m						O	O	O	O
NUMBER OF LINKS	R-x\L\N						O	O	O	O
LINK NAME	R-x\LNM-n						O	O	O	O
LINK SOURCE STREAM NAME	R-x\LSNM-n						O	O	O	O
LINK SOURCE STREAM NUMBER	R-x\LSSN-n						O	O	O	O
LINK DESTINATION DRIVE NUMBER	R-x\LDDN-n						O	O	O	O
LINK DESTINATION VOLUME NUMBER	R-x\LDVN-n						O	O	O	O
NUMBER OF ETHERNET PUBLISHING LINKS	R-x\EPL\N								O	O
LINK NAME	R-x\EPL\LNM-n								O	O
LINK SOURCE STREAM NAME	R-x\EPL\LSNM-n								O	O
LINK SOURCE STREAM NUMBER	R-x\EPL\LSSN-n								O	O
LINK DESTINATION ETHERNET INTERFACE IP ADDRESS	R-x\EPL\LDEIP-n								O	O
LINK DESTINATION ETHERNET INTERFACE PORT ADDRESS	R-x\EPL\LDEPA-n								O	O
USER DEFINED CHANNEL ID	R-x\UD\TK1					X	O	O	O	O
RECORDING EVENTS ENABLED	R-x\EV\E		X	X	X	X	O	O	O	O
RECORDING EVENTS CHANNEL ID	R-x\EV\TK1					X	O	O	O	O
NUMBER OF RECORDING EVENTS	R-x\EV\N		X	X	X	X	O	O	O	O
RECORDER INTERNAL EVENTS ENABLED	R-x\EV\IEE				X	X	O	O	O	O

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
EVENT ID	R-x\EV\ID-n		X	X	X	X	O	O	O	O
EVENT DESCRIPTION	R-x\EV\D-n		X	X	X	X	O	O	O	O
EVENT DATA PROCESSING ENABLED	R-x\EV\EDP-n					X	X	X	X	X
EVENT TYPE	R-x\EV\T-n		X	X	X	X	O	O	O	O
EVENT PRIORITY	R-x\EV\P-n		X	X	X	X	O	O	O	O
EVENT CAPTURE MODE	R-x\EV\CM-n			X	X	X	O	O	O	O
EVENT INITIAL CAPTURE	R-x\EV\IC-n			X	X	X	O	O	O	O
RECORDING EVENT LIMIT COUNT	R-x\EV\LC-n		X	X	X	X	O	O	O	O
EVENT MEASUREMENT SOURCE	R-x\EV\MS-n		X	X	X	X	O	O	O	O
EVENT MEASUREMENT NAME	R-x\EV\MN-n		X	X	X	X	O	O	O	O
EVENT PROCESSING MEASUREMENT DATA LINK NAME	R-x\EV\DLN-n					X	X	X	X	X
NUMBER OF MEASUREMENTS TO PROCESS	R-x\EV\PM\N-n					X	X	X	X	X
MEASUREMENT NAME TO PROCESS	R-x\EV\PM\MN-n-m					X	X	X	X	X
PRE-EVENT PROCESSING DURATION	R-x\EV\PM\PRE-n-m					X	X	X	X	X
POST-EVENT PROCESSING DURATION	R-x\EV\PM\PST-n-m					X	X	X	X	X
RECORDING INDEX ENABLED	R-x\IDX\E		X	X	X	X	O	O	O	O
RECORDING INDEX CHANNEL ID	R-x\IDX\TK1					X	O	O	O	O
RECORDING INDEX TYPE	R-x\IDX\IT		X	X	X	X	O	O	O	O
INDEX TIME VALUE	R-x\IDX\ITV		X	X	X	X	O	O	O	O
INDEX COUNT VALUE	R-x\IDX\ICV		X	X	X	X	O	O	O	O
MESSAGE MONITOR RECORD CONTROL ENABLED	R-x\MRC\E				X	X	X	X	X	X
CHANNEL ID NUMBER	R-x\MRC\ID				X	X	X	X	X	X
MESSAGE RECORD CONTROL TYPE	R-x\MRC\RCT				X	X	X	X	X	X
STOP-PAUSE COMMAND WORD	R-x\MRC\SPM				X	X	X	X	X	X
START-RESUME COMMAND WORD	R-x\MRC\SRM				X	X	X	X	X	X
TRACK NUMBER/ CHANNEL ID	R-x\TK1-n	X	X	X	X	X	R	R	R	R
RECORDING TECHNIQUE	R-x\TK2-n	X	X	X	X	X	X	X	X	X
INPUT STREAM DERANDOMIZATION	R-x\IDDR-n					X	X	X	X	X
DATA SOURCE ID	R-x\DSI-n	X	X	X	X	X	R	R	R	R
DATA DIRECTION	R-x\TK3-n	X	X	X	X	X	X	X	X	X
RECORDER PHYSICAL CHANNEL NUMBER	R-x\TK4-n			X	X	X	R	R	R	R
CHANNEL ENABLE	R-x\CHE-n	X	X	X	X	X	R	R	R	R
DATA SOURCE TYPE	R-x\DST-n	X								
CHANNEL DATA TYPE	R-x\CDT-n	X	X	X	X	X	R	R	R	R
DATA LINK NAME	R-x\BDLN-n	X	X							
PCM DATA LINK NAME	R-x\PDLN-n	X	X							
CHANNEL DATA LINK NAME	R-x\CDLN-n			X	X	X	R	R	R	R
SECONDARY HEADER TIME FORMAT	R-x\SHTF-n						O	O	O	O
PCM DATA TYPE FORMAT	R-x\PDTF-n			X	X	X	O	O	O	O
DATA PACKING OPTION	R-x\PDP-n	X	X	X	X	X	O	O	O	O

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
RECORDER POLARITY SETTING	R-x\RPS-n								O	O
INPUT CLOCK EDGE	R-x\ICE-n			X	X	X	O	O	O	O
INPUT SIGNAL TYPE	R-x\IST-n			X	X	X	O	O	O	O
INPUT THRESHOLD	R-x\ITH-n			X	X	X	O	O	O	O
INPUT TERMINATION	R-x\ITM-n			X	X	X	O	O	O	O
PCM VIDEO TYPE FORMAT	R-x\PTF-n	X	X	X	X	X	O	O	O	O
PCM RECORDER-REPRODUCER MINOR FRAME FILTERING ENABLED	R-x\MFF\E-n				X	X	O	O	O	O
PCM POST PROCESS OVERWRITE AND FILTERING ENABLED	R-x\POF\E-n				X	X	O	O	O	O
PCM POST PROCESS OVERWRITE AND FILTERING TYPE	R-x\POF\T-n				X	X	O	O	O	O
MINOR FRAME FILTERING DEFINITION TYPE	R-x\MFF\FDT-n						P	P	P	P
NUMBER OF MINOR FRAME FILTERING DEFINITIONS	R-x\MFF\N-n						P	P	P	P
FILTERED MINOR FRAME NUMBER	R-x\MFF\MFN-n-m						P	P	P	P
NUMBER OF SELECTED MEASUREMENT OVERWRITE DEFINITIONS	R-x\SMF\N-n						O	O	O	O
SELECTED MEASUREMENT NAME	R-x\SMF\SMN-n-m						O	O	O	O
MEASUREMENT OVERWRITE TAG	R-x\SMF\MFOT-n-m						O	O	O	O
MIL-STD-1553 BUS DATA TYPE FORMAT	R-x\BTF-n			X	X	X	O	O	O	O
MIL-STD-1553 RECORDER ? REPRODUCER FILTERING ENABLED	R-x\MRF\E-n				X	X	O	O	O	O
MIL-STD-1553 POST PROCESS OVERWRITE AND FILTERING ENABLED	R-x\MOF\T-n				X	X	O	O	O	O
MIL-STD-1553 MESSAGE FILTERING DEFINITION TYPE	R-x\MFD\FDT-n				X	X	X	X	X	X
NUMBER OF MESSAGE FILTERING DEFINITIONS	R-x\MFD\N-n				X	X	X	X	X	X
MESSAGE NUMBER	R-x\MFD\MID-n-m						X	X	X	X
MESSAGE TYPE	R-x\MFD\MT-n-m				X	X	X	X	X	X
COMMAND WORD ENTRY	R-x\CWE-n-m					X	X	X	X	X
COMMAND WORD	R-x\CMD-n-m					X	X	X	X	X
REMOTE TERMINAL ADDRESS	R-x\MFD\TRA-n-m				X	X	X	X	X	X
TRANSMIT/RECEIVE MODE	R-x\MFD\TRM-n-m				X	X	X	X	X	X
SUBTERMINAL ADDRESS	R-x\MFD\STA-n-m				X	X	X	X	X	X
DATA WORD COUNT/MODE CODE	R-x\MFD\DWC-n-m				X	X	X	X	X	X
RECEIVE COMMAND WORD ENTRY	R-x\RCWE-n-m					X	X	X	X	X
RECEIVE COMMAND WORD	R-x\RCMD-n-m					X	X	X	X	X
RT/RT REMOTE TERMINAL ADDRESS	R-x\MFD\RTRA-n-m				X	X	X	X	X	X
RT/RT SUBTERMINAL ADDRESS	R-x\MFD\RSTA-n-m				X	X	X	X	X	X
RT/RT DATA WORD COUNT	R-x\MFD\RDWC-n-m				X	X	X	X	X	X
NUMBER OF SELECTED MEASUREMENT OVERWRITE DEFINITIONS	R-x\BME\N-n						O	O	O	O
SELECTED MEASUREMENT NAME	R-x\BME\SMN-n-m						O	O	O	O
MEASUREMENT OVERWRITE TAG	R-x\BME\MFOT-n-m						O	O	O	O
ANALOG DATA TYPE FORMAT	R-x\ATF-n			X	X	X	O	O	O	O

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
NUMBER OF ANALOG CHANNELS/PKT	R-x\ACH\N-n	X	X	X	X	X	O	O	O	O
DATA PACKING OPTION	R-x\ADP-n	X	X	X	X	X	O	O	O	O
SAMPLE RATE	R-x\ASR-n	X	X	X	X	X	O	O	O	O
SUB CHANNEL ENABLED	R-x\AMCE-n-m							R	R	R
NUMBER OF SUB CHANNEL ENABLED	R-x\AMCN-n							R	R	R
MEASUREMENT NAME	R-x\AMN-n-m	X	X	X	X	X	O	O	O	O
DATA LENGTH	R-x\ADL-n-m	X	X	X	X	X	O	O	O	O
BIT MASK	R-x\AMSK-n-m	X	X	X	X	X	O	O	O	O
MEASUREMENT TRANSFER ORDER	R-x\AMTO-n-m	X	X	X	X	X	O	O	O	O
SAMPLE FACTOR	R-x\ASF-n-m	X	X	X	X	X	O	O	O	O
SAMPLE FILTER 3DB BANDWIDTH	R-x\ASBW-n-m			X	X	X	O	O	O	O
AC/DC COUPLING	R-x\ACP-n-m			X	X	X	O	O	O	O
RECORDER INPUT IMPEDANCE	R-x\AII-n-m			X	X	X	O	O	O	O
INPUT CHANNEL GAIN	R-x\AGI-n-m			X	X	X	O	O	O	O
INPUT FULL SCALE RANGE	R-x\AFSI-n-m			X	X	X	O	O	O	O
INPUT OFFSET VOLTAGE	R-x\AOVI-n-m			X	X	X	O	O	O	O
LSB VALUE	R-x\ALSV-n-m			X						
EUC SLOPE	R-x\AECS-n-m			X						
EUC OFFSET	R-x\AECO-n-m			X						
EUC UNITS	R-x\AECU-n-m			X						
FORMAT Note: Omitting R-x\AF-n-m in 2009 was an oversight.	R-x\AF-n-m			X		X	O	O	O	O
INPUT TYPE	R-x\AIT-n-m			X	X	X	O	O	O	O
AUDIO	R-x\AV-n-m			X	X	X	O	O	O	O
AUDIO FORMAT	R-x\AVF-n-m			X	X	X	O	O	O	O
DISCRETE DATA TYPE FORMAT	R-x\DTF-n			X	X	X	O	O	O	O
DISCRETE MODE	R-x\DMOD-n	X	X	X	X	X	O	O	O	O
SAMPLE RATE	R-x\DSR-n	X	X	X	X	X	O	O	O	O
NUMBER OF DISCRETE MEASUREMENTS	R-x\NDM\N-n	X	X	X	X	X	O	O	O	O
MEASUREMENT NAME	R-x\DMN-n-m	X	X	X	X	X	O	O	O	O
BIT MASK	R-x\DMSK-n-m	X	X	X	X	X	O	O	O	O
MEASUREMENT TRANSFER ORDER	R-x\DMTO-n-m	X	X	X	X	X	O	O	O	O
ARINC 429 BUS DATA TYPE FORMAT	R-x\ABTF-n			X	X	X	O	O	O	O
NUMBER OF ARINC 429 SUB-CHANNELS	R-x\NAS\N-n			X	X	X	O	O	O	O
ARINC 429 SUB-CHANNEL NUMBER	R-x\ASN-n-m			X	X	X	O	O	O	O
ARINC 429 SUB-CHANNEL NAME	R-x\ANM-n-m			X	X	X	O	O	O	O
VIDEO DATA TYPE FORMAT	R-x\VTF-n			X	X	X	O	O	O	O
MPEG-2 CHANNEL XON2 FORMAT	R-x\VXF-n			X	X	X	O	O	O	O
VIDEO SIGNAL TYPE	R-x\VST-n			X	X	X	O	O	O	O
VIDEO SIGNAL FORMAT TYPE	R-x\VSF-n			X	X	X	O	O	O	O
VIDEO CONSTANT BIT RATE	R-x\CBR-n			X	X	X	O	O	O	O

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
VIDEO VARIABLE PEAK BIT RATE	R-x\VBR-n			X	X	X	O	O	O	O
VIDEO ENCODING DELAY	R-x\VED-n	X	X	X	X	X	O	O	O	O
OVERLAY ENABLED	R-x\VCO\OE-n				X	X	X	X	X	X
OVERLAY X POSITION	R-x\VCO\X-n				X	X	X	X	X	X
OVERLAY Y POSITION	R-x\VCO\Y-n				X	X	X	X	X	X
OVERLAY EVENT TOGGLE ENABLED	R-x\VCO\OET-n				X	X	X	X	X	X
OVERLAY FORMAT	R-x\VCO\OLF-n				X	X	X	X	X	X
OVERLAY BACKGROUND	R-x\VCO\OBG-n				X	X	X	X	X	X
ANALOG AUDIO CHANNEL INPUT LEFT	R-x\ASI\ASL-n				X	X	X	X	X	X
ANALOG AUDIO CHANNEL INPUT RIGHT	R-x\ASI\ASR-n				X	X	X	X	X	X
VIDEO DATA ALIGNMENT	R-x\VDA-n				X	X	O	O	O	O
TIME DATA TYPE FORMAT	R-x\TTF-n			X	X	X	R	R	R	R
TIME FORMAT	R-x\TFMT-n	X	X	X	X	X	R	R	R	R
TIME SOURCE	R-x\TSRC-n		X	X	X	X	R	R	R	R
IMAGE DATA TYPE FORMAT	R-x\ITF-n			X	X	X	O	O	O	O
STILL IMAGE TYPE	R-x\SIT-n			X	X	X	O	O	O	O
DYNAMIC IMAGE FORMAT	R-x\DIF-n						O	O	O	O
IMAGE TIME STAMP MODE	R-x\ITSM-n						O	O	O	O
DYNAMIC IMAGE ACQUISITION MODE	R-x\DIAM-n						O	O	O	O
IMAGE FRAME RATE	R-x\IFR-n						O	O	O	O
PRE-TRIGGER FRAMES	R-x\PTG-n						X	X	X	X
TOTAL FRAMES	R-x\TOTF-n						X	X	X	X
EXPOSURE TIME	R-x\EXP-n						X	X	X	X
SENSOR ROTATION	R-x\ROT-n						X	X	X	X
SENSOR GAIN VALUE	R-x\SGV-n						X	X	X	X
SENSOR AUTO GAIN	R-x\SAG-n						X	X	X	X
SENSOR WIDTH	R-x\ISW-n						O	O	O	O
SENSOR HEIGHT	R-x\ISH-n						O	O	O	O
MAX IMAGE WIDTH	R-x\MIW-n						O	O	O	O
MAX IMAGE HEIGHT	R-x\MIH-n						O	O	O	O
IMAGE WIDTH	R-x\IW-n						O	O	O	O
IMAGE HEIGHT	R-x\IH-n						O	O	O	O
IMAGE OFFSET X	R-x\IOX-n						O	O	O	O
IMAGE OFFSET Y	R-x\IOY-n						O	O	O	O
LINE PITCH	R-x\ILP-n						X	X	X	X
BINNING HORIZONTAL	R-x\IBH-n						X	X	X	X
BINNING VERTICAL	R-x\IBV-n						X	X	X	X
DECIMATION HORIZONTAL	R-x\IDH-n						X	X	X	X
DECIMATION VERTICAL	R-x\IDV-n						X	X	X	X
REVERSE X	R-x\IRX-n						X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
REVERSE Y	R-x\IRY-n						X	X	X	X
PIXEL DYNAMIC RANGE MINIMUM	R-x\IPMN-n						X	X	X	X
PIXEL DYNAMIC RANGE MAXIMUM	R-x\IPMX-n						X	X	X	X
TEST IMAGE TYPE	R-x\TIT-n						X	X	X	X
UART DATA TYPE FORMAT	R-x\UTF-n			X	X	X	O	O	O	O
NUMBER OF UART SUB-CHANNELS	R-x\NUS\N-n			X	X	X	O	O	O	O
UART SUB-CHANNEL NUMBER	R-x\USCN-n-m			X	X	X	O	O	O	O
UART SUB-CHANNEL NAME	R-x\UCNM-n-m			X	X	X	O	O	O	O
UART SUB-CHANNEL BAUD RATE	R-x\UCR-n-m				X	X	O	O	O	O
UART SUB-CHANNEL BITS PER WORD	R-x\UCB-n-m				X	X	O	O	O	O
UART SUB-CHANNEL PARITY	R-x\UCP-n-m				X	X	O	O	O	O
UART SUB-CHANNEL STOP BIT	R-x\UCS-n-m				X	X	O	O	O	O
UART SUB-CHANNEL INTERFACE	R-x\UCIN-n-m				X	X	X	X	X	X
UART SUB-CHANNEL BLOCK SIZE	R-x\UCBS-n-m				X	X	X	X	X	X
UART SUB-CHANNEL SYNC WORD LENGTH	R-x\UCSL-n-m				X	X	X	X	X	X
UART SUB-CHANNEL BLOCK SYNC VALUE	R-x\UCSV-n-m				X	X	X	X	X	X
UART SUB-CHANNEL BLOCK RATE	R-x\UCBR-n-m				X	X	X	X	X	X
MESSAGE DATA TYPE FORMAT	R-x\MTF-n			X	X	X	O	O	O	O
NUMBER OF MESSAGE SUB-CHANNELS	R-x\NMS\N-n			X	X	X	O	O	O	O
MESSAGE SUB-CHANNEL NUMBER	R-x\MSCN-n-m			X	X	X	O	O	O	O
MESSAGE SUB-CHANNEL NAME	R-x\MCNM-n-m			X	X	X	O	O	O	O
IEEE-1394 DATA TYPE FORMAT	R-x\IETF-n			X	X	X	O	O	O	O
PARALLEL DATA TYPE FORMAT	R-x\PLTF-n			X	X	X	O	O	O	O
ETHERNET DATA TYPE FORMAT	R-x\ENTF-n			X	X	X	O	O	O	O
NUMBER OF ETHERNET NETWORKS	R-x\NNET\N-n				X	X	O	O	O	O
ETHERNET NETWORK NUMBER	R-x\ENBR-n-m				X	X	O	O	O	O
ETHERNET NETWORK NAME	R-x\ENAM-n-m				X	X	O	O	O	O
TSPI/CTS DATA TYPE FORMAT	R-x\TDTF-n					X	O	O	O	O
CAN BUS DATA TYPE FORMAT	R-x\CBTF-n						O	O	O	O
NUMBER OF CAN BUS SUB-CHANNELS	R-x\NCB\N-n						O	O	O	O
CAN BUS SUB-CHANNEL NUMBER	R-x\CBN-n-m						O	O	O	O
CAN BUS SUB-CHANNEL NAME	R-x\CBM-n-m						O	O	O	O
CAN BUS BIT RATE	R-x\CBBS-n-m						O	O	O	O
FIBRE CHANNEL DATA TYPE FORMAT	R-x\FCTF-n							O	O	O
FIBRE CHANNEL SPEED	R-x\FCSP-n							O	O	O
OUTPUT STREAM NAME	R-x\OSNM-n							X	X	X
STREAM ID	R-x\SID-n							X	X	X
CONFIGURATION HASH RATE	R-x\HRATE-n							X	X	X
CONFIGURATION PACKET RATE	R-x\CRATE-n							X	X	X
NUMBER OF REFERENCE TRACKS	R-x\RT\N	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
TRACK NUMBER	R-x\RT1-n	X	X	X	X	X	X	X	X	X
REFERENCE FREQUENCY	R-x\RT2-n	X	X	X	X	X	X	X	X	X
COMMENTS	R-x\COM	X	X	X	X	X	O	O	O	O
DATA SOURCE ID	M-x\ID	X	X	X	X	X	X	X	X	X
SIGNAL STRUCTURE TYPE	M-x\BB1	X	X	X	X	X	X	X	X	X
MODULATION SENSE	M-x\BB2	X	X	X	X	X	X	X	X	X
COMPOSITE LPF BANDWIDTH	M-x\BB3	X	X	X	X	X	X	X	X	X
BASEBAND SIGNAL TYPE	M-x\BSG1	X	X	X	X	X	X	X	X	X
BANDWIDTH	M-x\BSF1	X	X	X	X	X	X	X	X	X
TYPE	M-x\BSF2	X	X	X	X	X	X	X	X	X
DATA LINK NAME	M-x\BB\DLN	X	X	X	X	X	X	X	X	X
MEASUREMENT NAME	M-x\BB\MN	X	X	X	X	X	X	X	X	X
NUMBER OF SUBCARRIERS	M-x\SCO\N	X	X	X	X	X	X	X	X	X
NUMBER OF SCOs	M-x\SI\N	X	X	X	X	X	X	X	X	X
SCO NUMBER	M-x\SI1-n	X	X	X	X	X	X	X	X	X
SCO #n DATA TYPE	M-x\SI2-n	X	X	X	X	X	X	X	X	X
MODULATION SENSE	M-x\SI3-n	X	X	X	X	X	X	X	X	X
BANDWIDTH	M-x\SIF1-n	X	X	X	X	X	X	X	X	X
TYPE	M-x\SIF2-n	X	X	X	X	X	X	X	X	X
DATA LINK NAME	M-x\SI\DLN-n	X	X	X	X	X	X	X	X	X
MEASUREMENT NAME	M-x\SI\MN-n	X	X	X	X	X	X	X	X	X
OTHER	M-x\SO	X	X	X	X	X	X	X	X	X
REFERENCE CHANNEL	M-x\RC	X	X	X	X	X	X	X	X	X
COMMENTS	M-x\COM	X	X	X	X	X	X	X	X	X
DATA LINK NAME	P-d\DLN	X	X	X	X	X	O	O	O	O
PCM CODE	P-d\D1	X	X	X	X	X	O	O	O	O
BIT RATE	P-d\D2	X	X	X	X	X	O	O	O	O
ENCRYPTED	P-d\D3	X	X	X	X	X	X	X	X	X
POLARITY	P-d\D4	X	X	X	X	X	O	O	O	O
AUTO-POLARITY CORRECTION	P-d\D5	X	X	X	X	X	X	X	X	X
DATA DIRECTION	P-d\D6	X	X	X	X	X	X	X	X	X
DATA RANDOMIZED	P-d\D7	X	X	X	X	X	O	O	O	O
RANDOMIZER LENGTH	P-d\D8	X	X	X	X	X	O	O	O	O
TYPE FORMAT	P-d\TF	X	X	X	X	X	O	O	O	O
COMMON WORD LENGTH	P-d\F1	X	X	X	X	X	P	P	P	P
WORD TRANSFER ORDER	P-d\F2	X	X	X	X	X	P	P	P	P
PARITY	P-d\F3	X	X	X	X	X	P	P	P	P
PARITY TRANSFER ORDER	P-d\F4	X	X	X	X	X	X	X	X	X
CRC	P-d\CRC						X	X	X	X
CRC CHECK WORD STARTING BIT	P-d\CRCCB						X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
CRC DATA START BIT	P-d\CRCDB						X	X	X	X
CRC DATA NUMBER OF BITS	P-d\CRCDN						X	X	X	X
NUMBER OF MINOR FRAMES IN MAJOR FRAME	P-d\MF\N	X	X	X	X	X	P	P	P	P
NUMBER OF WORDS IN A MINOR FRAME	P-d\MF1	X	X	X	X	X	P	P	P	P
NUMBER OF BITS IN A MINOR FRAME	P-d\MF2	X	X	X	X	X	P	P	P	P
SYNC TYPE	P-d\MF3	X	X	X	X	X	X	X	X	X
LENGTH	P-d\MF4	X	X	X	X	X	P	P	P	P
PATTERN	P-d\MF5	X	X	X	X	X	P	P	P	P
IN SYNC CRITERIA	P-d\SYNC1	X	X	X	X	X	X	X	X	X
SYNC PATTERN CRITERIA	P-d\SYNC2	X	X	X	X	X	X	X	X	X
NUMBER OF DISAGREES	P-d\SYNC3	X	X	X	X	X	X	X	X	X
SYNC PATTERN CRITERIA	P-d\SYNC4	X	X	X	X	X	X	X	X	X
FILL BITS	P-d\SYNC5						X	X	X	X
NUMBER OF UNIQUE WORD SIZES	P-d\MFW\N								X	X
WORD NUMBER	P-d\MFW1-n	X	X	X	X	X	P	P	P	P
NUMBER OF BITS IN WORD	P-d\MFW2-n	X	X	X	X	X	P	P	P	P
NUMBER OF SUBFRAME ID COUNTERS	P-d\ISF\N	X	X	X	X	X	P	P	P	P
SUBFRAME ID COUNTER NAME	P-d\ISF1-n	X	X	X	X	X	P	P	P	P
SUBFRAME SYNC TYPE	P-d\ISF2-n	X	X	X	X	X	P	P	P	P
SUBFRAME ID COUNTER LOCATION	P-d\IDC1-n	X	X	X	X	X	P	P	P	P
ID COUNTER WORD LENGTH <wrap tip>(Omitted in 106-13 and -15?)</wrap>	P-d\IDC2-n	X	X	X	X	X				
ID COUNTER MSB STARTING BIT LOCATION	P-d\IDC3-n	X	X	X	X	X	P	P	P	P
ID COUNTER LENGTH	P-d\IDC4-n	X	X	X	X	X	P	P	P	P
ID COUNTER TRANSFER ORDER	P-d\IDC5-n	X	X	X	X	X	P	P	P	P
ID COUNTER INITIAL VALUE	P-d\IDC6-n	X	X	X	X	X	P	P	P	P
INITIAL COUNT SUBFRAME NUMBER	P-d\IDC7-n	X	X	X	X	X	P	P	P	P
ID COUNTER END VALUE	P-d\IDC8-n	X	X	X	X	X	P	P	P	P
END COUNT SUBFRAME NUMBER	P-d\IDC9-n	X	X	X	X	X	P	P	P	P
COUNT DIRECTION	P-d\IDC10-n	X	X	X	X	X	P	P	P	P
NUMBER OF SUBFRAMES	P-d\SF\N-n	X	X	X	X					
SUBFRAME NAME	P-d\SF1-n-m	X	X	X	X					
SUPERCOM	P-d\SF2-n-m	X	X	X	X					
LOCATION DEFINITION	P-d\SF3-n-m	X	X	X	X					
SUBFRAME LOCATION	P-d\SF4-n-m-w	X	X	X	X					
INTERVAL	P-d\SF5-n-m	X	X	X	X					
SUBFRAME DEPTH	P-d\SF6-n-m	X	X	X	X					
MINOR FRAME FILTERING DEFINITION TYPE	P-d\MFF\FDT				X	X				
NUMBER OF MINOR FRAME FILTERING DEFINITIONS	P-d\MFF\N				X	X				
FILTERED MINOR FRAME NUMBER	P-d\MFF\MFN-n				X	X				
NUMBER OF ASYNCHRONOUS EMBEDDED FORMATS	P-d\AEF\N	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
DATA LINK NAME	P-d\AEF\DLN-n	X	X	X	X	X	X	X	X	X
SUPERCOM	P-d\AEF1-n	X	X	X	X	X	X	X	X	X
LOCATION DEFINITION	P-d\AEF2-n	X	X	X	X	X	X	X	X	X
LOCATION	P-d\AEF3-n-w	X	X	X	X	X	X	X	X	X
INTERVAL	P-d\AEF4-n	X	X	X	X	X	X	X	X	X
WORD LENGTH	P-d\AEF5-n-w	X	X	X	X	X	X	X	X	X
MASK	P-d\AEF6-n-w	X	X	X	X	X	X	X	X	X
SUBCOMMUTATED	P-d\AEF7-n-w					X	X	X	X	X
START FRAME	P-d\AEF8-n-w-m					X	X	X	X	X
FRAME INTERVAL	P-d\AEF9-n-w-m					X	X	X	X	X
END FRAME	P-d\AEF10-n-w-m					X	X	X		
LOCATION	P-d\FFI1	X	X	X	X	X	X	X	X	X
MASK	P-d\FFI2	X	X	X	X	X	X	X	X	X
NUMBER OF MEASUREMENT LISTS	P-d\MLC\N	X	X	X	X	X	X	X	X	X
FFI PATTERN	P-d\MLC1-n	X	X	X	X	X	X	X	X	X
MEASUREMENT LIST NAME	P-d\MLC2-n	X	X	X	X	X	X	X	X	X
NUMBER OF FORMATS	P-d\FSC\N	X	X	X	X	X	X	X	X	X
FFI PATTERN	P-d\FSC1-n	X	X	X	X	X	X	X	X	X
DATA LINK ID	P-d\FSC2-n	X	X	X	X	X	X	X	X	X
NUMBER OF TAGS	P-d\ALT\N	X	X	X	X	X	X	X	X	X
NUMBER OF BITS IN TAG	P-d\ALT1	X	X	X	X	X	X	X	X	X
NUMBER OF BITS IN DATA WORD	P-d\ALT2	X	X	X	X	X	X	X	X	X
FIRST TAG LOCATION	P-d\ALT3	X	X	X	X	X	X	X	X	X
SEQUENCE	P-d\ALT4	X	X	X	X	X	X	X	X	X
FORMATS	P-d\ADM\N	X	X	X	X	X	X	X	X	X
DATA MERGE NAME	P-d\ADM\DMN-n	X	X	X	X	X	X	X	X	X
MASK AND PATTERN	P-d\ADM\MP-n					X	X	X	X	X
OVERHEAD MASK	P-d\ADM\OHM-n					X	X	X	X	X
FRESH DATA PATTERN	P-d\ADM\FDP-n					X	X	X	X	X
DATA OVERFLOW PATTERN	P-d\ADM\DOP-n					X	X	X	X	X
STALE DATA PATTERN	P-d\ADM\SDP-n					X	X	X	X	X
USER DEFINED PATTERN	P-d\ADM\UDP-n					X	X	X	X	X
SUPERCOM	P-d\ADM1-n	X	X	X	X	X	X	X	X	X
LOCATION DEFINITION	P-d\ADM2-n	X	X	X	X	X	X	X	X	X
LOCATION	P-d\ADM3-n-w	X	X	X	X	X	X	X	X	X
INTERVAL	P-d\ADM4-n	X	X	X	X	X	X	X	X	X
DATA LENGTH	P-d\ADM5-n	X	X	X	X	X	X	X	X	X
MSB LOCATION	P-d\ADM6-n	X	X	X	X	X	X	X	X	X
PARITY	P-d\ADM7-n	X	X	X	X	X	X	X	X	X
SUBCOMMUTATED	P-d\ADM8-n-w					X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
START FRAME	P-d\ADM9-n-w-m					X	X	X	X	X
FRAME INTERVAL	P-d\ADM10-n-w-m					X	X	X	X	X
END FRAME	P-d\ADM11-n-w-m					X	X	X		
CHAPTER 7 NUMBER OF SEGMENTS	P-d\C7\N								X	X
CHAPTER 7 FIRST WORD OF SEGMENT	P-d\C7FW-n								X	X
CHAPTER 7 NUMBER OF PCM WORDS IN SEGMENT	P-d\C7NW-n								X	X
COMMENTS	P-d\COM	X	X	X	X	X	X	X	X	X
DATA LINK NAME	D-x\DLN	X	X	X	X	X	X	X	X	X
NUMBER OF MEASUREMENT LISTS	D-x\ML\N	X	X	X	X	X	X	X	X	X
MEASUREMENT LIST NAME	D-x\MLN-y	X	X	X	X	X	X	X	X	X
NUMBER OF SELECTED MEASUREMENT FILTERING AND OVERWRITE DEFINITIONS	D-x\SMF\N-y				X					
NUM OF MEASUREMENT FILTERING AND OVERWRITE DEFINITIONS	D-x\SMF\N					X				
SELECTED MEASUREMENT NAME	D-x\SMF\SMN-y-n				X					
SELECTED MEASUREMENT NAME	D-x\SMF\SMN-y					X				
MEASUREMENT FILTER-OVERWRITE TAG	D-x\SMF\MFOT-y					X				
NUMBER OF MEASURANDS	D-x\MN\N-y	X	X	X	X	X	X	X	X	X
MEASUREMENT NAME	D-x\MN-y-n	X	X	X	X	X	X	X	X	X
PARITY	D-x\MN1-y-n	X	X	X	X	X	X	X	X	X
PARITY TRANSFER ORDER	D-x\MN2-y-n	X	X	X	X	X	X	X	X	X
MEASUREMENT TRANSFER ORDER	D-x\MN3-y-n	X	X	X	X	X	X	X	X	X
MEASUREMENT FILTER-OVERWRITE TAG	D-x\MFOT-y-n				X					
MEASUREMENT LOCATION TYPE	D-x\LT-y-n	X	X	X	X	X	X	X	X	X
MINOR FRAME LOCATION	D-x\MF-y-n	X	X	X	X					
BIT MASK	D-x\MFM-y-n	X	X	X	X					
NUMBER OF MINOR FRAME LOCATIONS	D-x\MFS\N-y-n	X	X	X	X					
LOCATION DEFINITION	D-x\MFS1-y-n	X	X	X	X					
LOCATION IN MINOR FRAME	D-x\MFS2-y-n	X	X	X	X					
BIT MASK	D-x\MFS3-y-n	X	X	X	X					
INTERVAL	D-x\MFS4-y-n	X	X	X	X					
MINOR FRAME LOCATION	D-x\MFSW-y-n-e	X	X	X	X					
BIT MASK	D-x\MFSM-y-n-e	X	X	X	X					
NUMBER OF FRAGMENTS	D-x\FMF\N-y-n	X	X	X	X					
MEASUREMENT WORD LENGTH	D-x\FMF1-y-n	X	X	X	X					
LOCATION DEFINITION	D-x\FMF2-y-n	X	X	X	X					
LOCATION IN MINOR FRAME	D-x\FMF3-y-n	X	X	X	X					
BIT MASK	D-x\FMF4-y-n	X	X	X	X					
INTERVAL	D-x\FMF5-y-n	X	X	X	X					
MINOR FRAME LOCATION	D-x\FMF6-y-n-e	X	X	X	X					
BIT MASK	D-x\FMF7-y-n-e	X	X	X	X					

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
FRAGMENT TRANSFER ORDER	D-x\FMF8-y-n-e	X	X	X	X					
FRAGMENT POSITION	D-x\FMF9-y-n-e	X	X	X	X					
SUBFRAME NAME	D-x\SF1-y-n	X	X	X	X					
LOCATION IN SUBFRAME	D-x\SF2-y-n	X	X	X	X					
BIT MASK	D-x\SFM-y-n	X	X	X	X					
NUMBER OF SUBFRAME LOCATIONS	D-x\SFS\N-y-n	X	X	X	X					
SUBFRAME NAME	D-x\SFS1-y-n	X	X	X	X					
LOCATION DEFINITION	D-x\SFS2-y-n	X	X	X	X					
LOCATION IN SUBFRAME	D-x\SFS3-y-n	X	X	X	X					
BIT MASK	D-x\SFS4-y-n	X	X	X	X					
INTERVAL	D-x\SFS5-y-n	X	X	X	X					
SUBFRAME LOCATION	D-x\SFS6-y-n-e	X	X	X	X					
BIT MASK	D-x\SFS7-y-n-e	X	X	X	X					
NUMBER OF FRAGMENTS	D-x\FSF\N-y-n	X	X	X	X					
MEASUREMENT WORD LENGTH	D-x\FSF1-y-n	X	X	X	X					
NUMBER OF SUBFRAMES	D-x\FSF2\N-y-n	X	X	X	X					
SUBFRAME NAME	D-x\FSF3-y-n-m	X	X	X	X					
LOCATION DEFINITION	D-x\FSF4-y-n-m	X	X	X	X					
LOCATION IN SUBFRAME	D-x\FSF5-y-n-m	X	X	X	X					
BIT MASK	D-x\FSF6-y-n-m	X	X	X	X					
INTERVAL	D-x\FSF7-y-n-m	X	X	X	X					
SUBFRAME LOCATION	D-x\FSF8-y-n-m-e	X	X	X	X					
BIT MASK	D-x\FSF9-y-n-m-e	X	X	X	X					
FRAGMENT TRANSFER ORDER	D-x\FSF10-y-n-m-e	X	X	X	X					
FRAGMENT POSITION	D-x\FSF11-y-n-m-e	X	X	X	X					
SUBFRAME ID COUNTER NAME	D-x\IDCN-y-n					X	X	X	X	X
NUMBER OF MEASUREMENT LOCATIONS	D-x\MML\N-y-n	X	X	X	X	X	X	X	X	X
NUMBER OF FRAGMENTS	D-x\MNF\N-y-n-m	X	X	X	X	X	X	X	X	X
MEASUREMENT WORD LENGTH	D-x\MWL-y-n-m	X	X	X	X	X	X	X		
WORD POSITION	D-x\WP-y-n-m-e	X	X	X	X	X	X	X	X	X
WORD INTERVAL	D-x\WI-y-n-m-e	X	X	X	X	X	X	X	X	X
END WORD POSITION	D-x\EWP-y-n-m-e				X	X	X	X		
FRAME POSITION	D-x\FP-y-n-m-e	X	X	X	X	X	X	X	X	X
FRAME INTERVAL	D-x\FI-y-n-m-e	X	X	X	X	X	X	X	X	X
END FRAME POSITION	D-x\EFP-y-n-m-e				X	X	X	X		
BIT MASK	D-x\WFM-y-n-m-e	X	X	X	X	X	X	X	X	X
FRAGMENT TRANSFER ORDER	D-x\WFT-y-n-m-e	X	X	X	X	X	X	X	X	X
FRAGMENT POSITION	D-x\WFP-y-n-m-e	X	X	X	X	X	X	X	X	X
SAMPLING MODE	D-x\SS-y-n								X	X
SAMPLE ON	D-x\SON-y-n								X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
SAMPLE ON MEASUREMENT NAME	D-x\SMN-y-n								X	X
NUMBER OF WORD FRAME SAMPLES	D-x\SS\N-y-n								X	X
SAMPLE ON WORD	D-x\SS1-y-n-s								X	X
SAMPLE ON FRAME	D-x\SS2-y-n-s								X	X
NUMBER OF TAG DEFINITIONS	D-x\TD\N-y-n					X	X	X	X	X
MEASUREMENT WORD LENGTH	D-x\TD1-y-n					X				
TAG NUMBER	D-x\TD2-y-n-m					X	X	X	X	X
BIT MASK	D-x\TD3-y-n-m					X	X	X	X	X
FRAGMENT TRANSFER ORDER	D-x\TD4-y-n-m					X	X	X	X	X
FRAGMENT POSITION	D-x\TD5-y-n-m					X	X	X	X	X
NUMBER OF PARENT MEASUREMENTS	D-x\REL\N-y-n						X	X	X	X
PARENT MEASUREMENT	D-x\REL1-y-n-m						X	X	X	X
BIT MASK	D-x\REL2-y-n-m						X	X	X	X
FRAGMENT TRANSFER ORDER	D-x\REL3-y-n-m						X	X	X	X
FRAGMENT POSITION	D-x\REL4-y-n-m						X	X	X	X
COMMENTS	D-x\COM	X	X	X	X	X	X	X	X	X
DATA LINK NAME	B-x\DLN	X	X	X	X	X	X	X	X	X
TEST ITEM	B-x\TA	X	X	X	X	X	X	X	X	X
BUS PARITY	B-x\BP			X	X	X	X	X	X	X
NUMBER OF BUSES	B-x\NBS\N	X	X	X	X	X	X	X	X	X
BUS NUMBER	B-x\BID-i	X	X	X	X	X	X	X	X	X
BUS NAME	B-x\BNA-i	X	X	X	X	X	X	X	X	X
BUS TYPE	B-x\BT-i	X	X	X	X	X	X	X	X	X
USER DEFINED WORD 1 MEASUREMENT	B-x\UMN1-i			X	X	X	X	X	X	X
PARITY	B-x\U1P-i				X	X	X	X	X	X
PARITY TRANSFER ORDER	B-x\U1PT-i				X	X	X	X	X	X
BIT MASK	B-x\U1M-i				X	X	X	X	X	X
TRANSFER ORDER	B-x\U1T-i				X	X	X	X	X	X
USER DEFINED WORD 2 MEASUREMENT	B-x\UMN2-i			X	X	X	X	X	X	X
PARITY	B-x\U2P-i				X	X	X	X	X	X
PARITY TRANSFER ORDER	B-x\U2PT-i				X	X	X	X	X	X
BIT MASK	B-x\U2M-i				X	X	X	X	X	X
TRANSFER ORDER	B-x\U2T-i				X	X	X	X	X	X
USER DEFINED WORD 3 MEASUREMENT	B-x\UMN3-i			X	X	X	X	X	X	X
PARITY	B-x\U3P-i				X	X	X	X	X	X
PARITY TRANSFER ORDER	B-x\U3PT-i				X	X	X	X	X	X
BIT MASK	B-x\U3M-i				X	X	X	X	X	X
TRANSFER ORDER	B-x\U3T-i				X	X	X	X	X	X
NUMBER OF TRACKS	B-x\TK\N-i	X	X	X	X	X	X	X	X	X
TRACK SEQUENCE	B-x\TS-i-k	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
NUMBER OF MESSAGES	B-x\NMS\N-i	X	X	X	X	X	X	X	X	X
MESSAGE NUMBER	B-x\MID-i-n	X	X	X	X	X	X	X	X	X
MESSAGE NAME	B-x\MNA-i-n	X	X	X	X	X	X	X	X	X
COMMAND WORD ENTRY	B-x\CWE-i-n					X	X	X	X	X
COMMAND WORD	B-x\CMD-i-n					X	X	X	X	X
REMOTE TERMINAL NAME	B-x\TRN-i-n	X	X	X	X	X	X	X	X	X
REMOTE TERMINAL ADDRESS	B-x\TRA-i-n	X	X	X	X	X	X	X	X	X
SUBTERMINAL NAME	B-x\STN-i-n	X	X	X	X	X	X	X	X	X
SUBTERMINAL ADDRESS	B-x\STA-i-n	X	X	X	X	X	X	X	X	X
TRANSMIT/RECEIVE MODE	B-x\TRM-i-n	X	X	X	X	X	X	X	X	X
DATA WORD COUNT/MODE CODE	B-x\DWC-i-n	X	X	X	X	X	X	X	X	X
SPECIAL PROCESSING	B-x\SPR-i-n	X	X	X	X	X	X	X	X	X
MESSAGE FILTER-OVERWRITE TAG	B-x\BMG\FOT-i-n				X	X				
ARINC 429 LABEL	B-x\LBL-i-n	X	X	X	X	X	X	X	X	X
ARINC 429 SDI CODE	B-x\SDI-i-n	X	X	X	X	X	X	X	X	X
RECEIVE COMMAND WORD ENTRY	B-x\RCWE-i-n					X	X	X	X	X
RECEIVE COMMAND WORD	B-x\RCMD-i-n					X	X	X	X	X
REMOTE TERMINAL NAME	B-x\RTRN-i-n-m	X	X	X	X	X	X	X	X	X
REMOTE TERMINAL ADDRESS	B-x\RTRA-i-n-m	X	X	X	X	X	X	X	X	X
SUBTERMINAL NAME	B-x\RSTN-i-n-m	X	X	X	X	X	X	X	X	X
SUBTERMINAL ADDRESS	B-x\RSTA-i-n-m	X	X	X	X	X	X	X	X	X
DATA WORD COUNT	B-x\RDWC-i-n-m	X	X	X	X	X	X	X	X	X
MODE CODE DESCRIPTION	B-x\MCD-i-n	X	X	X	X	X	X	X	X	X
MODE CODE DATA WORD DESCRIPTION	B-x\MCW-i-n	X	X	X	X	X	X	X	X	X
NUMBER OF MEASURANDS	B-x\MN\N-i-n	X	X	X	X	X	X	X	X	X
MEASUREMENT NAME	B-x\MN-i-n-p	X	X	X	X	X	X	X	X	X
MEASUREMENT TYPE	B-x\MT-i-n-p			X	X	X	X	X	X	X
PARITY	B-x\MN1-i-n-p	X	X	X	X	X	X	X	X	X
PARITY TRANSFER ORDER	B-x\MN2-i-n-p	X	X	X	X	X	X	X	X	X
MEASUREMENT FILTER-OVERWRITE TAG	B-x\BME\FOT-i-n-p				X	X				
NUMBER OF MEASUREMENT LOCATIONS	B-x\NML\N-i-n-p	X	X	X	X	X	X	X	X	X
MESSAGE WORD NUMBER	B-x\MWN-i-n-p-e	X	X	X	X	X	X	X	X	X
BIT MASK	B-x\MBM-i-n-p-e	X	X	X	X	X	X	X	X	X
TRANSFER ORDER	B-x\MTO-i-n-p-e	X	X	X	X	X	X	X	X	X
FRAGMENT POSITION	B-x\MFP-i-n-p-e	X	X	X	X	X	X	X	X	X
COMMENTS	B-x\COM	X	X	X	X	X	X	X	X	X
DATA LINK NAME	S-d\DLN		X	X	X	X	X	X	X	X
ATTACHED SYNCHRONIZATION MARKER	S-x\ASM		X	X						
FRAME ERROR CONTROL FIELD FLAG	S-x\FEF		X	X						
CONVOLUTIONAL ERROR DETECTION AND CORRECTION FLAG	S-x\CEF		X	X						

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
TRANSFER FRAME LENGTH	S-x\TFL		X	X						
TEST ARTICLE ID	S-d\TA		X	X	X	X	X	X	X	X
VIRTUAL CHANNEL ID	S-x\VID		X	X						
OPERATIONAL CONTROL FIELD FLAG	S-x\OCF		X	X						
TRANSFER FRAME SECONDARY HEADER FLAG	S-x\SHF		X	X						
SYNC FLAG	S-x\OSF		X	X						
TRANSFER FRAME SECONDARY HEADER LENGTH	S-x\SHL		X	X						
NUMBER OF MEASUREMENTS	S-x\TNMS\N		X	X						
MEASUREMENT NAME	S-x\TMN-n		X	X						
PARITY	S-x\TPAR-n		X	X						
PARITY TRANSFER ORDER	S-x\TPTO-n		X	X						
NUMBER OF MEASUREMENT LOCATIONS	S-x\TNML\N-n		X	X						
WORD POSITION	S-x\TWP-n-m		X	X						
WORD LENGTH	S-x\TWL-n-m		X	X						
BIT MASK	S-x\TBM-n-m		X	X						
TRANSFER ORDER	S-x\TTO-n-m		X	X						
FRAGMENT POSITION	S-x\TFP-n-m		X	X						
TRANSFER FRAME DATA FIELD LENGTH	S-x\DFL		X	X						
NUMBER OF SOURCE PACKETS	S-x\SP\N		X	X						
PACKET SECONDARY HEADER FLAG	S-x\PSHF-n		X	X						
APPLICATION PROCESS ID	S-x\APID-n		X	X						
PACKET DATA LENGTH	S-x\PDL-n		X	X						
PACKET SECONDARY HEADER LENGTH	S-x\PSHL-n		X	X						
NUMBER OF MEASUREMENTS	S-x\HNMS\N-n		X	X						
MEASUREMENT NAME	S-x\HMN-n-m		X	X						
PARITY	S-x\HPAR-n-m		X	X						
PARITY TRANSFER ORDER	S-x\HPTO-n-m		X	X						
NUMBER OF MEASUREMENT LOCATIONS	S-x\HNML\N-n-m		X	X						
WORD POSITION	S-x\HWP-n-m-e		X	X						
WORD LENGTH	S-x\HWL-n-m-e		X	X						
BIT MASK	S-x\HBM-n-m-e		X	X						
TRANSFER ORDER	S-x\HTO-n-m-e		X	X						
FRAGMENT POSITION	S-x\HFP-n-m-e		X	X						
NUMBER OF MEASUREMENTS	S-x\SNMS\N-n		X	X						
MEASUREMENT NAME	S-x\SMN-n-m		X	X						
PARITY	S-x\SPAR-n-m		X	X						
PARITY TRANSFER ORDER	S-x\SPTO-n-m		X	X						
NUMBER OF MEASUREMENT LOCATIONS	S-x\SNML\N-n-m		X	X						
WORD POSITION	S-x\SWP-n-m-e		X	X						
WORD LENGTH	S-x\SWL-n-m-e		X	X						

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
BIT MASK	S-x\SBM-n-m-e		X	X						
TRANSFER ORDER	S-x\STO-n-m-e		X	X						
FRAGMENT POSITION	S-x\SFP-n-m-e		X	X						
NUMBER OF STREAMS	S-d\NS\N				X	X	X	X	X	X
STREAM NAME	S-d\SNA-i				X	X	X	X	X	X
MESSAGE DATA TYPE	S-d\MDT-i				X	X	X	X	X	X
MESSAGE DATA LAYOUT	S-d\MDL-i				X	X	X	X	X	X
MESSAGE ELEMENT SIZE	S-d\MES-i				X	X	X	X	X	X
MESSAGE ID LOCATION	S-d\MIDL-i				X	X	X	X	X	X
MESSAGE LENGTH	S-d\MLEN-i				X	X	X	X	X	X
MESSAGE DELIMITER	S-d\MDEL-i				X	X	X	X	X	X
MESSAGE DELIMITER LENGTH	S-d\MDLEN-i				X	X	X	X	X	X
FIELD DELIMITER	S-d\FDEL-i				X	X	X	X	X	X
DATA ORIENTATION	S-d\DO-i				X	X	X	X	X	X
NUMBER OF MESSAGES	S-d\NMS\N-i				X	X	X	X	X	X
MESSAGE ID	S-d\MID-i-n				X	X	X	X	X	X
MESSAGE DESCRIPTION	S-d\MNA-i-n				X	X	X	X	X	X
NUMBER OF FIELDS	S-d\NFLDS\N-i-n				X	X	X	X	X	X
FIELD NUMBER	S-d\FNUM-i-n-m				X	X	X	X	X	X
FIELD START	S-d\FPOS-i-n-m				X	X	X	X	X	X
FIELD LENGTH	S-d\FLEN-i-n-m				X	X	X	X	X	X
NUMBER OF MEASURANDS	S-d\MN\N-i-n				X	X	X	X	X	X
MEASUREMENT NAME	S-d\MN-i-n-p				X	X	X	X	X	X
PARITY	S-d\MN1-i-n-p				X	X	X	X	X	X
PARITY TRANSFER ORDER	S-d\MN2-i-n-p				X	X	X	X	X	X
DATA TYPE	S-d\MBFM-i-n-p				X	X	X	X	X	X
FLOATING POINT FORMAT	S-d\MFPF-i-n-p				X	X	X	X	X	X
DATA ORIENTATION	S-d\MDO-i-n-p				X	X	X	X	X	X
NUMBER OF MEASUREMENT LOCATIONS	S-d\NML\N-i-n-p				X	X	X	X	X	X
MESSAGE FIELD NUMBER	S-d\MFN-i-n-p-e				X	X	X	X	X	X
BIT MASK	S-d\MBM-i-n-p-e				X	X	X	X	X	X
TRANSFER ORDER	S-d\MTO-i-n-p-e				X	X	X	X	X	X
FRAGMENT POSITION	S-d\MFP-i-n-p-e				X	X	X	X	X	X
COMMENTS	S-x\COM-n		X	X						
COMMENTS	S-d\COM		X	X	X	X	X	X	X	X
DATA LINK NAME	A-x\DLN	X		X	X	X				
INPUT CODE	A-x\A1	X		X	X	X				
POLARITY	A-x\A2	X		X	X	X				
SYNC PATTERN TYPE	A-x\A3	X		X	X	X				
SYNC PATTERN (OTHER)	A-x\A4	X		X	X	X				

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
CHANNEL RATE	A-x\A5	X		X	X	X				
CHANNELS PER FRAME	A-x\A\N	X		X	X	X				
NUMBER OF MEASURANDS	A-x\A\MN\N	X		X	X	X				
0% SCALE CHANNEL NUMBER	A-x\RC1	X		X	X	X				
50% SCALE CHANNEL NUMBER	A-x\RC2	X		X	X	X				
FULL SCALE CHANNEL NUMBER	A-x\RC3	X		X	X	X				
NUMBER OF SUBFRAMES	A-x\SF\N	X		X	X	X				
SUBFRAME n LOCATION	A-x\SF1-n	X		X	X	X				
SUBFRAME n SYNCHRONIZATION	A-x\SF2-n	X		X	X	X				
SUBFRAME n SYNCHRONIZATION PATTERN	A-x\SF3-n	X		X	X	X				
MEASUREMENT NAME	A-x\MN1-n	X		X	X	X				
SUBCOM	A-x\MN2-n	X		X	X	X				
SUPERCOM	A-x\MN3-n	X		X	X	X				
CHANNEL NUMBER	A-x\LCW-n-s	X		X	X	X				
SUBFRAME CHANNEL NUMBER	A-x\LCN-n-s-r	X		X	X	X				
COMMENTS	A-x\COM	X		X	X	X				
NAME	C-d\DCN	X	X	X	X	X	X	X	X	X
TYPE	C-d\TRD1	X	X	X	X	X	X	X	X	X
MODEL NUMBER	C-d\TRD2	X	X	X	X	X	X	X	X	X
SERIAL NUMBER	C-d\TRD3	X	X	X	X	X	X	X	X	X
SECURITY CLASSIFICATION	C-d\TRD4	X	X	X	X	X	X	X	X	X
ORIGINATION DATE	C-d\TRD5	X	X	X	X	X	X	X	X	X
REVISION NUMBER	C-d\TRD6	X	X	X	X	X	X	X	X	X
ORIENTATION	C-d\TRD7	X	X	X	X	X	X	X	X	X
NAME	C-d\POC1	X	X	X	X	X	X	X	X	X
AGENCY	C-d\POC2	X	X	X	X	X	X	X	X	X
ADDRESS	C-d\POC3	X	X	X	X	X	X	X	X	X
TELEPHONE	C-d\POC4	X	X	X	X	X	X	X	X	X
DESCRIPTION	C-d\MN1	X	X	X	X	X	X	X	X	X
MEASUREMENT ALIAS	C-d\MNA	X	X	X	X	X	X	X	X	X
EXCITATION VOLTAGE	C-d\MN2	X	X	X	X	X	X	X	X	X
ENGINEERING UNITS	C-d\MN3	X	X	X	X	X	X	X	X	X
LINK TYPE	C-d\MN4	X	X	X	X	X	X	X	X	X
BINARY FORMAT	C-d\BFM	X	X	X	X	X	X	X	X	X
FLOATING POINT FORMAT	C-d\FPF	X	X	X	X	X	X	X	X	X
NUMBER OF BITS	C-d\BWT\N			X	X	X	X	X	X	X
BIT NUMBER	C-d\BWTB-n			X	X	X	X	X	X	X
BIT WEIGHT VALUE	C-d\BWTV-n			X	X	X	X	X	X	X
NUMBER OF POINTS	C-d\MC\N	X	X	X	X	X	X	X	X	X
STIMULUS	C-d\MC1-n	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
TELEMETRY VALUE	C-d\MC2-n	X	X	X	X	X	X	X	X	X
DATA VALUE	C-d\MC3-n	X	X	X	X	X	X	X	X	X
NUMBER OF AMBIENT CONDITIONS	C-d\MA\N	X	X	X	X	X	X	X	X	X
STIMULUS	C-d\MA1-n	X	X	X	X	X	X	X	X	X
TELEMETRY VALUE	C-d\MA2-n	X	X	X	X	X	X	X	X	X
DATA VALUE	C-d\MA3-n	X	X	X	X	X	X	X	X	X
FILTERING ENABLED	C-d\FEN									X
FILTERING DELAY	C-d\FDL									X
NUMBER OF FILTERS	C-d\F\N									X
FILTER TYPE	C-d\FTY-n									X
NUMBER OF POLES OR SAMPLES	C-d\FNPS-n									X
HIGH MEASUREMENT VALUE	C-d\MOT1	X	X	X	X	X	X	X	X	X
LOW MEASUREMENT VALUE	C-d\MOT2	X	X	X	X	X	X	X	X	X
HIGH ALERT LIMIT VALUE	C-d\MOT3	X	X	X	X	X	X	X	X	X
LOW ALERT LIMIT VALUE	C-d\MOT4	X	X	X	X	X	X	X	X	X
HIGH WARNING LIMIT VALUE	C-d\MOT5	X	X	X	X	X	X	X	X	X
LOW WARNING LIMIT VALUE	C-d\MOT6	X	X	X	X	X	X	X	X	X
INITIAL VALUE	C-d\MOT7						X	X	X	X
SAMPLE RATE	C-d\SR	X	X	X	X	X	X	X	X	X
DATE AND TIME RELEASED	C-d\CRT	X	X	X	X	X	X	X	X	X
CONVERSION TYPE	C-d\DCT	X	X	X	X	X	X	X	X	X
NUMBER OF SETS	C-d\PS\N	X	X	X	X	X	X	X	X	X
APPLICATION	C-d\PS1	X	X	X	X	X	X	X	X	X
ORDER OF FIT	C-d\PS2	X	X	X	X	X	X	X	X	X
TELEMETRY VALUE	C-d\PS3-n	X	X	X	X	X	X	X	X	X
ENGINEERING UNITS VALUE	C-d\PS4-n	X	X	X	X	X	X	X	X	X
ORDER OF CURVE FIT	C-d\CO\N	X	X	X	X	X	X	X	X	X
DERIVED FROM PAIR SET	C-d\CO1	X	X	X	X	X	X	X	X	X
COEFFICIENT (0)	C-d\CO	X	X	X	X	X	X	X	X	X
N-TH COEFFICIENT	C-d\CO-n	X	X	X	X	X	X	X	X	X
ORDER	C-d\NPC\N			X	X	X	X	X	X	X
DERIVED FROM PAIR SET	C-d\NPC1			X	X	X	X	X	X	X
COEFFICIENT (0)	C-d\NPC			X	X	X	X	X	X	X
N-TH COEFFICIENT	C-d\NPC-n			X	X	X	X	X	X	X
DEFINITION OF OTHER DATA CONVERSION	C-d\OTH	X	X	X	X	X	X	X	X	X
ALGORITHM TYPE	C-d\DPAT				X	X	X	X	X	X
ALGORITHM	C-d\DPA	X	X	X	X	X	X	X	X	X
TRIGGER MEASURAND	C-d\DPTM				X	X	X	X	X	X
NUMBER OF OCCURRENCES	C-d\DPNO				X	X	X	X	X	X
NUMBER OF INPUT MEASURANDS	C-d\DP\N	X	X	X	X	X	X	X	X	X

Table A-1. Supported TMATS Code Values

TMATS Attribute	Code	04	05	07	09	11	13	15	17	19
MEASURAND #N	C-d\DP-n	X	X	X	X	X	X	X	X	X
NUMBER OF INPUT CONSTANTS	C-d\DPC\N	X	X	X	X	X	X	X	X	X
CONSTANT #N	C-d\DPC-n	X	X	X	X	X	X	X	X	X
NUMBER OF EVENTS	C-d\DIC\N	X	X	X	X	X	X	X	X	X
NUMBER OF INDICATORS	C-d\DICI\N	X	X	X	X	X	X	X	X	X
CONVERSION DATA	C-d\DICC-n	X	X	X	X	X	X	X	X	X
PARAMETER EVENT DEFINITION	C-d\DICP-n	X	X	X	X	X	X	X	X	X
PCM TIME WORD FORMAT	C-d\PTM	X	X	X	X	X	X	X	X	X
1553 TIME WORD FORMAT	C-d\BTM	X	X	X	X	X	X	X	X	X
ENCODING METHOD	C-d\VOI\E	X	X	X	X	X	X	X	X	X
DESCRIPTION	C-d\VOI\D	X	X	X	X	X	X	X	X	X
ENCODING METHOD	C-d\VID\E	X	X	X	X	X	X	X	X	X
DESCRIPTION	C-d\VID\D	X	X	X	X	X	X	X	X	X
COMMENTS	C-d\COM	X	X	X	X	X	X	X	X	X

This page intentionally left blank.

Appendix B

Selected Source Code Files

A software implementation of the IRIG 106 Chapter 9 and Chapter 10 standard is available from www.irig106.org and is open source, meaning it is freely available in source code form. It is written in ANSI C, and can be compiled in GNU GCC as well as the various Microsoft Visual Studio compiler suites. Below are selected source files demonstrating important aspects of IRIG 106. These source files are also necessary for the example programs in subsequent appendices.

This code is under active development. Go to www.irig106.org for the latest version of this source code as well as additional source code files.

Included below are the following source files.

irig106ch10.h irig106ch10.c	Structures, macro definitions, and procedures for opening, reading/ writing, and moving through an IRIG 106 data file.
i106_time.h i106_time.c	Structures, macro definitions, and procedures for handling time in general
i106_decode_time.h i106_decode_time.c	Structures, macro definitions, and procedures for decoding Time Data packets
i106_data_stream.h i106_data_stream.c	Structures, macro definitions, and procedures for decoding Ch 10 UDP data streaming.
config.h	Structures and macro definitions to provide portability across supported compiler suites.

Appendix A-1. irig106ch10.h

```
/****************************************************************************

  irig106ch10.h -

 Copyright (c) 2005 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

/** @file
IRIG 106 library core routines

*/

#ifndef _irig106ch10_h_
#define _irig106ch10_h_

#ifdef __cplusplus
namespace Irig106 {
extern "C" {
#endif

#include "config.h"

/*
 * Macros and definitions
 * ----------------------
 */


#if !defined(bTRUE)
#define bTRUE       ((int)(1==1))
#define bFALSE      ((int)(1==0))
#endif
```

```
#define MAX_HANDLES         100

#define IRIG106_SYNC        0xEB25

// Define the longest file path string size
#undef  MAX_PATH
#define MAX_PATH                        260

// Header and secondary header sizes
#define HEADER_SIZE         24
#define SEC_HEADER_SIZE     12

// Header packet flags
#define I106CH10_PFLAGS_CHKSUM_NONE       (uint8_t)0x00
#define I106CH10_PFLAGS_CHKSUM_8          (uint8_t)0x01
#define I106CH10_PFLAGS_CHKSUM_16         (uint8_t)0x02
#define I106CH10_PFLAGS_CHKSUM_32         (uint8_t)0x03
#define I106CH10_PFLAGS_CHKSUM_MASK       (uint8_t)0x03

#define I106CH10_PFLAGS_TIMEFMT_IRIG106   (uint8_t)0x00
#define I106CH10_PFLAGS_TIMEFMT_IEEE1588  (uint8_t)0x04
#define I106CH10_PFLAGS_TIMEFMT_Reserved1 (uint8_t)0x08
#define I106CH10_PFLAGS_TIMEFMT_Reserved2 (uint8_t)0x0C
#define I106CH10_PFLAGS_TIMEFMT_MASK      (uint8_t)0x0C

#define I106CH10_PFLAGS_OVERFLOW          (uint8_t)0x10
#define I106CH10_PFLAGS_TIMESYNCERR       (uint8_t)0x20
#define I106CH10_PFLAGS_IPTIMESRC         (uint8_t)0x40
#define I106CH10_PFLAGS_SEC_HEADER        (uint8_t)0x80

// Header data types
#define I106CH10_DTYPE_COMPUTER_0       (uint8_t)0x00
#define I106CH10_DTYPE_USER_DEFINED     (uint8_t)0x00
#define I106CH10_DTYPE_COMPUTER_1       (uint8_t)0x01
#define I106CH10_DTYPE_TMATS            (uint8_t)0x01
#define I106CH10_DTYPE_COMPUTER_2       (uint8_t)0x02
#define I106CH10_DTYPE_RECORDING_EVENT  (uint8_t)0x02
#define I106CH10_DTYPE_COMPUTER_3       (uint8_t)0x03
#define I106CH10_DTYPE_RECORDING_INDEX  (uint8_t)0x03
#define I106CH10_DTYPE_COMPUTER_4       (uint8_t)0x04
#define I106CH10_DTYPE_STREAMING_CONFIG (uint8_t)0x04
#define I106CH10_DTYPE_COMPUTER_5       (uint8_t)0x05
#define I106CH10_DTYPE_COMPUTER_6       (uint8_t)0x06
#define I106CH10_DTYPE_COMPUTER_7       (uint8_t)0x07
#define I106CH10_DTYPE_PCM_FMT_0        (uint8_t)0x08
#define I106CH10_DTYPE_PCM_FMT_1        (uint8_t)0x09
#define I106CH10_DTYPE_PCM              (uint8_t)0x09   // Depricated
#define I106CH10_DTYPE_IRIG_TIME        (uint8_t)0x11
#define I106CH10_DTYPE_NETWORK_TIME     (uint8_t)0x12
#define I106CH10_DTYPE_1553_FMT_1       (uint8_t)0x19
#define I106CH10_DTYPE_1553_FMT_2       (uint8_t)0x1A   // 16PP194 Bus
#define I106CH10_DTYPE_16PP194          (uint8_t)0x1A   // 16PP194 Bus
#define I106CH10_DTYPE_ANALOG           (uint8_t)0x21
#define I106CH10_DTYPE_DISCRETE         (uint8_t)0x29
#define I106CH10_DTYPE_MESSAGE          (uint8_t)0x30
#define I106CH10_DTYPE_ARINC_429_FMT_0  (uint8_t)0x38
#define I106CH10_DTYPE_VIDEO_FMT_0      (uint8_t)0x40
#define I106CH10_DTYPE_VIDEO_FMT_1      (uint8_t)0x41
#define I106CH10_DTYPE_VIDEO_FMT_2      (uint8_t)0x42
#define I106CH10_DTYPE_VIDEO_FMT_3      (uint8_t)0x43
#define I106CH10_DTYPE_VIDEO_FMT_4      (uint8_t)0x44
#define I106CH10_DTYPE_IMAGE_FMT_0      (uint8_t)0x48
```

```
#define I106CH10_DTYPE_IMAGE_FMT_1      (uint8_t)0x49
#define I106CH10_DTYPE_IMAGE_FMT_2      (uint8_t)0x4A
#define I106CH10_DTYPE_UART_FMT_0       (uint8_t)0x50
#define I106CH10_DTYPE_1394_FMT_0       (uint8_t)0x58
#define I106CH10_DTYPE_1394_FMT_1       (uint8_t)0x59
#define I106CH10_DTYPE_PARALLEL_FMT_0   (uint8_t)0x60
#define I106CH10_DTYPE_ETHERNET_FMT_0   (uint8_t)0x68
#define I106CH10_DTYPE_ETHERNET_FMT_1   (uint8_t)0x69
#define I106CH10_DTYPE_ETHERNET_A664    (uint8_t)0x69
#define I106CH10_DTYPE_TSPI_FMT_0       (uint8_t)0X70
#define I106CH10_DTYPE_TSPI_FMT_1       (uint8_t)0X71
#define I106CH10_DTYPE_TSPI_FMT_2       (uint8_t)0X72
#define I106CH10_DTYPE_CAN              (uint8_t)0X78
#define I106CH10_DTYPE_FC_FMT_0         (uint8_t)0X79
#define I106CH10_DTYPE_FC_FMT_1         (uint8_t)0X7A


/// Error return codes
typedef enum EnStatus
    {
    I106_OK                 =  0,   ///< Everything okey dokey
    I106_OPEN_ERROR         =  1,   ///< Fatal problem opening for read or write
    I106_OPEN_WARNING       =  2,   ///< Non-fatal problem opening for read or write
    I106_EOF                =  3,   ///< End of file encountered
    I106_BOF                =  4,   //
    I106_READ_ERROR         =  5,   ///< Error reading data from file
    I106_WRITE_ERROR        =  6,   ///< Error writing data to file
    I106_MORE_DATA          =  7,   ///< More read data available
    I106_SEEK_ERROR         =  8,   ///< Unable to seek to positino
    I106_WRONG_FILE_MODE    =  9,   ///< Operation compatible with file open mode
    I106_NOT_OPEN           = 10,   ///< File not open for reading or writing
    I106_ALREADY_OPEN       = 11,   ///< File already open
    I106_BUFFER_TOO_SMALL   = 12,   ///< User buffer too small to hold data
    I106_NO_MORE_DATA       = 13,   ///< No more data to read
    I106_NO_FREE_HANDLES    = 14,   ///< Too many files open
    I106_INVALID_HANDLE     = 15,   ///< Passed file handle doesn't point to an open
file
    I106_TIME_NOT_FOUND     = 16,   ///< No valid time packet found
    I106_HEADER_CHKSUM_BAD  = 17,   ///< Invalid header checksum
    I106_NO_INDEX           = 18,   ///< No index found
    I106_UNSUPPORTED        = 19,   ///< Unsupported operation
    I106_BUFFER_OVERRUN     = 20,   ///< Data exceeds buffer size
    I106_INDEX_NODE         = 21,   ///< Returned decoded node message
    I106_INDEX_ROOT         = 22,   ///< Returned decoded root message
    I106_INDEX_ROOT_LINK    = 23,   ///< Returned decoded link to next root (i.e. last
root)
    I106_INVALID_DATA       = 24,   ///< Packet data is invalid for some reason
    I106_INVALID_PARAMETER  = 25    ///< Passed parameter is invalid
    } EnI106Status;

/// Data file open mode
typedef enum I106ChMode
    {
    I106_CLOSED             = 0,
    I106_READ               = 1,    ///< Open an existing file for reading
    I106_OVERWRITE          = 2,    ///< Create a new file or overwrite an exising
file
    I106_APPEND             = 3,    ///< Append data to the end of an existing file
    I106_READ_IN_ORDER      = 4,    ///< Open an existing file for reading in time
order
    I106_READ_NET_STREAM    = 5,    ///< Open network data stream for reading
    I106_WRITE_NET_STREAM   = 6,    ///< Open network data stream for writing
    } EnI106Ch10Mode;
```

```
/// Read state is used to keep track of the next expected data file structure
typedef enum FileState
    {
    enClosed        = 0,
    enWrite         = 1,
    enReadUnsynced  = 2,
    enReadHeader    = 3,
    enReadData      = 4,
    enReadNetStream = 5,
    } EnFileState;

/// Index sort state
typedef enum SortStatus
    {
    enUnsorted   = 0,
    enSorted     = 1,
    enSortError  = 2,
    } EnSortStatus;


/*
 * Data structures
 * ---------------
 */

#if defined(_MSC_VER)
#pragma pack(push)
#pragma pack(1)
#endif

/// IRIG 106 header and optional secondary header data structure
typedef PUBLIC struct SuI106Ch10Header_S
    {
    uint16_t      uSync;                ///< Packet Sync Pattern
    uint16_t      uChID;                ///< Channel ID
    uint32_t      ulPacketLen;          ///< Total packet length
    uint32_t      ulDataLen;            ///< Data length
    uint8_t       ubyHdrVer;            ///< Header Version
    uint8_t       ubySeqNum;            ///< Sequence Number
    uint8_t       ubyPacketFlags;       ///< PacketFlags
    uint8_t       ubyDataType;          ///< Data type
    uint8_t       aubyRefTime[6];       ///< Reference time
    uint16_t      uChecksum;            ///< Header Checksum
    uint8_t       abyTime[8];           ///< Time (start secondary header)
    uint16_t      uReserved;            //
    uint16_t      uSecChecksum;         ///< Secondary Header Checksum
#if !defined(__GNUC__)
    } SuI106Ch10Header;
#else
    } __attribute__ ((packed)) SuI106Ch10Header;
#endif

/// Structure for holding file index
typedef struct
    {
    int64_t     llOffset;               ///< File position byte offset
    int64_t     llTime;                 ///< Packet RTC at this offset
    } SuInOrderPacketInfo;


// Various file index array indexes
typedef struct
```

```
    {
    EnSortStatus            enSortStatus;
    SuInOrderPacketInfo   * asuIndex;
    int                     iArraySize;
    int                     iArrayUsed;
    int                     iArrayCurr;  // Current position in index array
    int64_t                 llNextReadOffset;
    int                     iNumSearchSteps;
    } SuInOrderIndex;

/// Data structure for IRIG 106 read/write handle
typedef struct
    {
    int                 bInUse;
    int                 iFile;
    char                szFileName[MAX_PATH];
    EnI106Ch10Mode      enFileMode;
    EnFileState         enFileState;
    SuInOrderIndex      suInOrderIndex;
    unsigned long       ulCurrPacketLen;
    unsigned long       ulCurrHeaderBuffLen;
    unsigned long       ulCurrDataBuffLen;
    unsigned long       ulCurrDataBuffReadPos;
    unsigned long       ulTotalBytesWritten;
    char                achReserve[128];
    } SuI106Ch10Handle;

#if defined(_MSC_VER)
#pragma pack(pop)
#endif


/*
 * Global data
 * -----------
 */

extern SuI106Ch10Handle  g_suI106Handle[MAX_HANDLES];


/*
 * Function Declaration
 * --------------------
 */

// Open / Close

/// Open a Chapter 10 file for reading or writing
EnI106Status I106_CALL_DECL
    enI106Ch10Open(
            int               * piI106Ch10Handle,
            const char          szOpenFileName[],
            EnI106Ch10Mode      enMode);

#if defined(IRIG_NETWORKING)
EnI106Status I106_CALL_DECL
    enI106Ch10OpenStreamRead(
            int               * piI106Ch10Handle,
            uint16_t            uPort);

EnI106Status I106_CALL_DECL
    enI106Ch10OpenStreamWrite(
            int               * piI106Ch10Handle,
```

```
            uint32_t            uIpAddress,
            uint16_t            uPort);
#endif

EnI106Status I106_CALL_DECL
    enI106Ch10Close(
            int                 iI106Handle);


// Read / Write
// ------------

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeader(int                iI106Ch10Handle,
                             SuI106Ch10Header  * psuI106Hdr);

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeaderFile(int                iHandle,
                                 SuI106Ch10Header * psuHeader);

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeaderInOrder(int                iHandle,
                                    SuI106Ch10Header * psuHeader);

EnI106Status I106_CALL_DECL
    enI106Ch10ReadPrevHeader(int                iI106Ch10Handle,
                             SuI106Ch10Header  * psuI106Hdr);

EnI106Status I106_CALL_DECL
    enI106Ch10ReadData(int                iI106Ch10Handle,
                       unsigned long      ulBuffSize,
                       void             * pvBuff);

EnI106Status I106_CALL_DECL
    enI106Ch10ReadDataFile(int                iHandle,
                           unsigned long      ulBuffSize,
                           void             * pvBuff);

EnI106Status I106_CALL_DECL
    enI106Ch10WriteMsg(int                  iI106Ch10Handle,
                       SuI106Ch10Header   * psuI106Hdr,
                       void               * pvBuff);


// Move file pointer
// -----------------

EnI106Status I106_CALL_DECL
    enI106Ch10FirstMsg(int iI106Ch10Handle);

EnI106Status I106_CALL_DECL
    enI106Ch10LastMsg(int iI106Ch10Handle);

EnI106Status I106_CALL_DECL
    enI106Ch10SetPos(int iI106Ch10Handle, int64_t llOffset);

EnI106Status I106_CALL_DECL
    enI106Ch10GetPos(int iI106Ch10Handle, int64_t * pllOffset);

// Utilities
// ---------

int I106_CALL_DECL
```

```
    iHeaderInit(SuI106Ch10Header * psuHeader,
                unsigned int       uChanID,
                unsigned int       uDataType,
                unsigned int       uFlags,
                unsigned int       uSeqNum);

int I106_CALL_DECL
    iGetHeaderLen(SuI106Ch10Header * psuHeader);

uint32_t I106_CALL_DECL
    uGetDataLen(SuI106Ch10Header * psuHeader);

uint16_t I106_CALL_DECL
    uCalcHeaderChecksum(SuI106Ch10Header * psuHeader);

uint16_t I106_CALL_DECL
    uCalcSecHeaderChecksum(SuI106Ch10Header * psuHeader);

char * szI106ErrorStr(EnI106Status enStatus);

uint32_t I106_CALL_DECL
    uCalcDataBuffReqSize(uint32_t uDataLen, int iChecksumType);

EnI106Status I106_CALL_DECL
    uAddDataFillerChecksum(SuI106Ch10Header * psuI106Hdr, unsigned char achData[]);

#ifdef __cplusplus
} // end namespace
} // end extern "C"
#endif

#endif
```

Appendix A-2. irig106ch10.c

```
/****************************************************************************

 irig106ch10.c -

 Copyright (c) 2005 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <errno.h>
#include <assert.h>

#if defined(__GNUC__)
#include <sys/io.h>
#else
#include <io.h>
#endif

#if defined(IRIG_NETWORKING) & !defined(_WIN32)
#include <sys/types.h>
#include <unistd.h>
#endif

#include "config.h"
#include "stdint.h"

#include "irig106ch10.h"
#include "i106_time.h"
```

```
#if defined(IRIG_NETWORKING)
#include "i106_data_stream.h"
#endif

#ifdef __cplusplus
namespace Irig106 {
#endif

/*
 * Macros and definitions
 * ----------------------
 */

#define BACKUP_SIZE     256


/*
 * Data structures
 * ---------------
 */


/*
 * Module data
 * -----------
 */

SuI106Ch10Handle  g_suI106Handle[MAX_HANDLES];
static int        m_bHandlesInited = bFALSE;

/*
 * Function Declaration
 * --------------------
 */

void InitHandles();
int GetNextHandle();

/* ------------------------------------------------------------------------ */

EnI106Status I106_CALL_DECL
    enI106Ch10Open(int             * piHandle,
                   const char        szFileName[],
                   EnI106Ch10Mode    enMode)
    {
    int                 iReadCnt;
    int                 iFlags;
    int                 iFileMode;
    uint16_t            uSignature;
    EnI106Status        enStatus;
    SuI106Ch10Header    suI106Hdr;

    // Initialize handle data if necessary
    InitHandles();

    // Get the next available handle
    *piHandle = GetNextHandle();
    if (*piHandle == -1)
        {
        return I106_NO_FREE_HANDLES;
        } // end if handle not found

    // Initialize some data
```

```
    g_suI106Handle[*piHandle].enFileState                = enClosed;
    g_suI106Handle[*piHandle].suInOrderIndex.enSortStatus = enUnsorted;

    // Get a copy of the file name
    strncpy (g_suI106Handle[*piHandle].szFileName, szFileName,
sizeof(g_suI106Handle[*piHandle].szFileName));
    g_suI106Handle[*piHandle].szFileName[sizeof(g_suI106Handle[*piHandle].szFileName)
- 1] = '\0';

    // Reset total bytes written
    g_suI106Handle[*piHandle].ulTotalBytesWritten = 0L;

/*** Read Mode ***/

    // Open for read
    if ((I106_READ == enMode) || (I106_READ_IN_ORDER == enMode))
        {

        //// Try to open file
#if defined(_MSC_VER)
        iFlags = O_RDONLY | O_BINARY;
#elif defined(__GNUC__)
        iFlags = O_RDONLY | O_LARGEFILE;
#else
        iFlags = O_RDONLY;
#endif
        g_suI106Handle[*piHandle].iFile = open(szFileName, iFlags, 0);
        if (g_suI106Handle[*piHandle].iFile == -1)
            {
            g_suI106Handle[*piHandle].bInUse = bFALSE;
            *piHandle = -1;
            return I106_OPEN_ERROR;
            }

        //// Check to make sure it is a valid IRIG 106 Ch 10 data file

        // Check for valid signature

        // If we couldn't even read the first 2 bytes then return error
        iReadCnt = read(g_suI106Handle[*piHandle].iFile, &uSignature, 2);
        if (iReadCnt != 2)
            {
            close(g_suI106Handle[*piHandle].iFile);
            g_suI106Handle[*piHandle].bInUse = bFALSE;
            *piHandle = -1;
            return I106_OPEN_ERROR;
            }

        // If the first word isn't the sync value then return error
        if (uSignature != IRIG106_SYNC)
            {
            close(g_suI106Handle[*piHandle].iFile);
            g_suI106Handle[*piHandle].bInUse = bFALSE;
            *piHandle = -1;
            return I106_OPEN_ERROR;
            }

        //// Reading data file looks OK so check some other stuff

        // Open OK and sync character OK so set read state to reflect this
        g_suI106Handle[*piHandle].enFileMode  = enMode;
        g_suI106Handle[*piHandle].enFileState = enReadHeader;
```

```
        // Make sure first packet is a config packet
//      fseek(g_suI106Handle[*piHandle].pFile, 0L, SEEK_SET);
        enI106Ch10SetPos(*piHandle, 0L);
        enStatus = enI106Ch10ReadNextHeaderFile(*piHandle, &suI106Hdr);
        if (enStatus != I106_OK)
            return I106_OPEN_WARNING;
        if (suI106Hdr.ubyDataType != I106CH10_DTYPE_COMPUTER_1)
            return I106_OPEN_WARNING;

//        // Make sure first dynamic data packet is a time packet
// THERE MAY BE MULTIPLE COMPUTER GENERATED PACKETS AT THE BEGINNING
//        fseek(psuHandle->pFile, suI106Hdr.ulPacketLen, SEEK_SET);
//        enStatus = enI106Ch10ReadNextHeaderFile(*piHandle, &suI106Hdr);
//        if (enStatus != I106_OK)
//            return I106_OPEN_WARNING;
//        if (suI106Hdr.ubyDataType != I106CH10_DTYPE_IRIG_TIME)
//            return I106_OPEN_WARNING;

        // Everything OK so get time and reset back to the beginning
//      fseek(g_suI106Handle[*piHandle].pFile, 0L, SEEK_SET);
        enI106Ch10SetPos(*piHandle, 0L);
        g_suI106Handle[*piHandle].enFileState = enReadHeader;
        g_suI106Handle[*piHandle].enFileMode  = enMode;

        // Do any presorting or indexing

        if (I106_READ_IN_ORDER == enMode)
        {
            g_suI106Handle[*piHandle].suInOrderIndex.iArrayUsed = 0;
            //vMakeInOrderIndex(*piHandle);
            g_suI106Handle[*piHandle].suInOrderIndex.iArrayCurr = 0;
        }

        } // end if read mode


/*** Overwrite Mode ***/

    // Open for overwrite
    else if (I106_OVERWRITE == enMode)
        {

        /// Try to open file
#if defined(_MSC_VER)
        iFlags    = O_WRONLY | O_CREAT | _O_TRUNC | O_BINARY;
        iFileMode = _S_IREAD | _S_IWRITE;
#elif defined(__GNUC__)
        iFlags    = O_WRONLY | O_CREAT | O_LARGEFILE;
        iFileMode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
#else
        iFlags    = O_WRONLY | O_CREAT;
        iFileMode = 0;
#endif
        g_suI106Handle[*piHandle].iFile = open(szFileName, iFlags, iFileMode);
        if (g_suI106Handle[*piHandle].iFile == -1)
            {
            g_suI106Handle[*piHandle].bInUse = bFALSE;
            *piHandle = -1;
            return I106_OPEN_ERROR;
            }

        // Open OK and write state to reflect this
        g_suI106Handle[*piHandle].enFileState = enWrite;
```

```
        g_suI106Handle[*piHandle].enFileMode  = enMode;
        } // end if read mode


/*** Any other mode is an error ***/

    else
        {
        g_suI106Handle[*piHandle].enFileState = enClosed;
        g_suI106Handle[*piHandle].enFileMode  = I106_CLOSED;
        g_suI106Handle[*piHandle].bInUse = bFALSE;
        *piHandle = -1;
        return I106_OPEN_ERROR;
        }

    return I106_OK;
    }



/* ---------------------------------------------------------------------- */

#if defined(IRIG_NETWORKING)

// Open a UDP network stream

EnI106Status I106_CALL_DECL
    enI106Ch10OpenStreamRead(int               * piHandle,
                             uint16_t            uPort)
    {
    EnI106Status    enStatus;

    // Initialize handle data if necessary
    InitHandles();

    // Get the next available handle
    *piHandle = GetNextHandle();
    if (*piHandle == -1)
        {
        return I106_NO_FREE_HANDLES;
        } // end if handle not found

    // Initialize some data
    g_suI106Handle[*piHandle].enFileState                 = enClosed;
    g_suI106Handle[*piHandle].suInOrderIndex.enSortStatus = enUnsorted;

    // Open the network data stream
    enStatus = enI106_OpenNetStreamRead(*piHandle, uPort);
    if (enStatus == I106_OK)
        {
        g_suI106Handle[*piHandle].enFileMode  = I106_READ_NET_STREAM;
        g_suI106Handle[*piHandle].enFileState = enReadHeader;
        }

    return enStatus;
    }


/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10OpenStreamWrite(
```

```
            int               * piHandle,
            uint32_t            uIpAddress,
            uint16_t            uPort)
    {
    EnI106Status    enStatus;

    // Initialize handle data if necessary
    InitHandles();

    // Get the next available handle
    *piHandle = GetNextHandle();
    if (*piHandle == -1)
        {
        return I106_NO_FREE_HANDLES;
        } // end if handle not found

    // Initialize some data
    g_suI106Handle[*piHandle].enFileState                = enClosed;

    // Open the network data stream
    enStatus = enI106_OpenNetStreamWrite(*piHandle, uIpAddress, uPort);
    if (enStatus == I106_OK)
        {
        g_suI106Handle[*piHandle].enFileMode  = I106_WRITE_NET_STREAM;
        }

    return enStatus;
    }


#endif

/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10Close(int iHandle)
    {

    // If handles have not been init'ed then bail
    if (m_bHandlesInited == bFALSE)
        return I106_NOT_OPEN;

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Handle different types of close
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        // Close the file
        default : // Default case is file
            // Make sure the file is really open
            if ((g_suI106Handle[iHandle].iFile   != -1) &&
                (g_suI106Handle[iHandle].bInUse  == bTRUE))
                close(g_suI106Handle[iHandle].iFile);
            break;

        // Close network data stream
        case I106_READ_NET_STREAM :
```

```
        case I106_WRITE_NET_STREAM :
#if defined(IRIG_NETWORKING)
            enI106_CloseNetStream(iHandle);
#endif
            break;

        } // end switch on file mode

    // Free index buffer and mark unsorted
    free(g_suI106Handle[iHandle].suInOrderIndex.asuIndex);
    g_suI106Handle[iHandle].suInOrderIndex.asuIndex        = NULL;
    g_suI106Handle[iHandle].suInOrderIndex.iArraySize      = 0;
    g_suI106Handle[iHandle].suInOrderIndex.iArrayUsed      = 0;
    g_suI106Handle[iHandle].suInOrderIndex.iNumSearchSteps = 0;
    g_suI106Handle[iHandle].suInOrderIndex.enSortStatus    = enUnsorted;

    // Reset some status variables
    g_suI106Handle[iHandle].iFile       = -1;
    g_suI106Handle[iHandle].bInUse      = bFALSE;
    g_suI106Handle[iHandle].enFileMode  = I106_CLOSED;
    g_suI106Handle[iHandle].enFileState = enClosed;

    return I106_OK;
    }



/* ---------------------------------------------------------------------- */

// Get the next header. Depending on how the file was opened for reading,
// call the appropriate routine.

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeader(int                iHandle,
                             SuI106Ch10Header * psuHeader)
    {
    EnI106Status    enStatus;

    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_READ_NET_STREAM :
        case I106_READ :
            enStatus = enI106Ch10ReadNextHeaderFile(iHandle, psuHeader);
            break;

        case I106_READ_IN_ORDER :
            if (g_suI106Handle[iHandle].suInOrderIndex.enSortStatus == enSorted)
                enStatus = enI106Ch10ReadNextHeaderInOrder(iHandle, psuHeader);
            else
                enStatus = enI106Ch10ReadNextHeaderFile(iHandle, psuHeader);
            break;

        default :
            enStatus = I106_WRONG_FILE_MODE;
            break;
        } // end switch on read mode

    return enStatus;
    }



/* ---------------------------------------------------------------------- */
```

```
// Get the next header in the file from the current position

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeaderFile(int                iHandle,
                                 SuI106Ch10Header * psuHeader)
    {
    int                 iReadCnt;
    int                 bReadHeaderWasOK;
    int64_t             llSkipSize;
    int64_t             llFileOffset;
    EnI106Status        enStatus;

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Check for invalid file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
        case I106_READ_IN_ORDER   :
        default                   :
            return I106_WRONG_FILE_MODE;
            break;

        case I106_READ_NET_STREAM :
        case I106_READ :
            break;
        } // end switch on read mode

    // Check file state
    switch (g_suI106Handle[iHandle].enFileState)
        {
        case enClosed :
            return I106_NOT_OPEN;
            break;

        case enWrite :
            return I106_WRONG_FILE_MODE;
            break;

        case enReadHeader :
            break;

        case enReadData :
            llSkipSize = g_suI106Handle[iHandle].ulCurrPacketLen -
                         g_suI106Handle[iHandle].ulCurrHeaderBuffLen -
                         g_suI106Handle[iHandle].ulCurrDataBuffReadPos;

            if (g_suI106Handle[iHandle].enFileMode != I106_READ_NET_STREAM)
                {
                enStatus = enI106Ch10GetPos(iHandle, &llFileOffset);
                if (enStatus != I106_OK)
```

```
                    return I106_SEEK_ERROR;

                llFileOffset += llSkipSize;

                enStatus = enI106Ch10SetPos(iHandle, llFileOffset);
                if (enStatus != I106_OK)
                    return I106_SEEK_ERROR;
                }
#if defined(IRIG_NETWORKING)
            else
                enI106_MoveReadPointer(iHandle, (long)llSkipSize);
                break;
#endif

        case enReadUnsynced :
            break;

        } // end switch on file state

    // Now we might be at the beginning of a header. Read what we think
    // is a header, check it, and keep reading if things don't look correct.
    while (bTRUE)
        {
        // Assume header is OK, only set false if not
        bReadHeaderWasOK = bTRUE;

        // Read the header
        if (g_suI106Handle[iHandle].enFileMode != I106_READ_NET_STREAM)
            iReadCnt = read(g_suI106Handle[iHandle].iFile, psuHeader, HEADER_SIZE);
#if defined(IRIG_NETWORKING)
        else
            iReadCnt = enI106_ReadNetStream(iHandle, psuHeader, HEADER_SIZE);
#endif

        // Keep track of how much header we've read
        g_suI106Handle[iHandle].ulCurrHeaderBuffLen = HEADER_SIZE;

        // If there was an error reading, figure out why
        if (iReadCnt != HEADER_SIZE)
            {
            g_suI106Handle[iHandle].enFileState = enReadUnsynced;
            if (iReadCnt == -1)
                return I106_READ_ERROR;
            else
                return I106_EOF;
            } // end if read error

        // Setup a one time loop to make it easy to break out if
        // there is an error encountered
        do
            {
            // Read OK, check the sync field
            if (psuHeader->uSync != IRIG106_SYNC)
                {
                g_suI106Handle[iHandle].enFileState = enReadUnsynced;
                bReadHeaderWasOK = bFALSE;
                break;
                }

            // Always check the header checksum
            if (psuHeader->uChecksum != uCalcHeaderChecksum(psuHeader))
                {
                // If the header checksum was bad then set to unsynced state
```

```
                // and return the error. Next time we're called we'll go
                // through lots of heroics to find the next header.
                if (g_suI106Handle[iHandle].enFileState != enReadUnsynced)
                    {
                    g_suI106Handle[iHandle].enFileState = enReadUnsynced;
                    return I106_HEADER_CHKSUM_BAD;
                    }
                bReadHeaderWasOK = bFALSE;
                break;
                }

            // MIGHT NEED TO CHECK HEADER VERSION HERE

            // Header seems OK at this point

            // Figure out if there is a secondary header
            if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_SEC_HEADER) != 0)
                {
                // Read the secondary header
                if (g_suI106Handle[iHandle].enFileMode != I106_READ_NET_STREAM)
                    iReadCnt = read(g_suI106Handle[iHandle].iFile, &psuHeader-
>aulTime[0], SEC_HEADER_SIZE);
#if defined(IRIG_NETWORKING)
                else
                    iReadCnt = enI106_ReadNetStream(iHandle, &psuHeader->aulTime[0],
SEC_HEADER_SIZE);
#endif

                // Keep track of how much header we've read
                g_suI106Handle[iHandle].ulCurrHeaderBuffLen += SEC_HEADER_SIZE;

                // If there was an error reading, figure out why
                if (iReadCnt != SEC_HEADER_SIZE)
                    {
                    g_suI106Handle[iHandle].enFileState = enReadUnsynced;
                    if (iReadCnt == -1)
                        return I106_READ_ERROR;
                    else
                        return I106_EOF;
                    } // end if read error

                // Always check the secondary header checksum now
                if (psuHeader->uSecChecksum != uCalcSecHeaderChecksum(psuHeader))
                    {
                    // If the header checksum was bad then set to unsynced state
                    // and return the error. Next time we're called we'll go
                    // through lots of heroics to find the next header.
                    if (g_suI106Handle[iHandle].enFileState != enReadUnsynced)
                        {
                        g_suI106Handle[iHandle].enFileState = enReadUnsynced;
                        return I106_HEADER_CHKSUM_BAD;
                        }
                    bReadHeaderWasOK = bFALSE;
                    break;
                    }

                } // end if secondary header

            } while (bFALSE); // end one time error testing loop

        // If read header was OK then break out
        if (bReadHeaderWasOK == bTRUE)
            break;
```

```
        // Read header was not OK so try again beyond previous read point
        if (g_suI106Handle[iHandle].enFileMode != I106_READ_NET_STREAM)
            {
            enStatus = enI106Ch10GetPos(iHandle, &llFileOffset);
            if (enStatus != I106_OK)
                return I106_SEEK_ERROR;

            llFileOffset = llFileOffset - g_suI106Handle[iHandle].ulCurrHeaderBuffLen
+ 1;

            enStatus = enI106Ch10SetPos(iHandle, llFileOffset);
            if (enStatus != I106_OK)
                return I106_SEEK_ERROR;
            }
#if defined(IRIG_NETWORKING)
        else
            {
            // Just dump the whole network receiver buffer and start fresh
            enI106_DumpNetStream(iHandle);
            }
#endif

        } // end while looping forever, looking for a good header

    // Save some data for later use
    g_suI106Handle[iHandle].ulCurrPacketLen       = psuHeader->ulPacketLen;
    g_suI106Handle[iHandle].ulCurrDataBuffLen     = uGetDataLen(psuHeader);
    g_suI106Handle[iHandle].ulCurrDataBuffReadPos = 0;
    g_suI106Handle[iHandle].enFileState           = enReadData;

    return I106_OK;
    } // end enI106Ch10ReadNextHeaderFile()



/* ------------------------------------------------------------------------ */

// Get the next header in time order from the file

EnI106Status I106_CALL_DECL
    enI106Ch10ReadNextHeaderInOrder(int                iHandle,
                                    SuI106Ch10Header * psuHeader)
    {

    SuInOrderIndex    * psuIndex = &g_suI106Handle[iHandle].suInOrderIndex;
    EnI106Status        enStatus;
    int64_t             llOffset;
    EnFileState         enSavedFileState;

    // If we're at the end of the list then we are at the end of the file
    if (psuIndex->iArrayCurr == psuIndex->iArrayUsed)
        return I106_EOF;

    // Save the read state going in
    enSavedFileState = g_suI106Handle[iHandle].enFileState;

    // Move file pointer to the proper, er, point
    llOffset = psuIndex->asuIndex[psuIndex->iArrayCurr].llOffset;
    enStatus = enI106Ch10SetPos(iHandle, llOffset);

    // Go ahead and get the next header
    enStatus = enI106Ch10ReadNextHeaderFile(iHandle, psuHeader);
```

```
    // If the state was unsynced before but is synced now, figure out where in the
    // index we are
    if ((enSavedFileState == enReadUnsynced) && (g_suI106Handle[iHandle].enFileState
!= enReadUnsynced))
        {
        enI106Ch10GetPos(iHandle, &llOffset);
        llOffset -= iGetHeaderLen(psuHeader);
        psuIndex->iArrayCurr = 0;
        while (psuIndex->iArrayCurr < psuIndex->iArrayUsed)
            {
            if (llOffset == psuIndex->asuIndex[psuIndex->iArrayCurr].llOffset)
                break;
            psuIndex->iArrayCurr++;
            }
        // if psuIndex->iArrayCurr == psuIndex->iArrayUsed then bad things happened
        }

    // Move array index to the next element
    psuIndex->iArrayCurr++;

    return enStatus;
    } // end enI106Ch10ReadNextHeaderInOrder()



/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10ReadPrevHeader(int                 iHandle,
                             SuI106Ch10Header  * psuHeader)
    {
    int                 bStillReading;
    int                 iReadCnt;
    int64_t             llCurrPos;
    EnI106Status        enStatus;

    uint64_t            llInitialBackup;
    int                 iBackupAmount;
    uint64_t            llNextBuffReadPos;
    uint8_t             abyScanBuff[BACKUP_SIZE+HEADER_SIZE];
    int                 iBuffIdx;
    uint16_t            uCheckSum;

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Check for invalid file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
        case I106_READ_IN_ORDER   : // HANDLE THE READ IN ORDER MODE!!!!
        case I106_READ_NET_STREAM :
```

```
        default                   :
            return I106_WRONG_FILE_MODE;
            break;

        case I106_READ :
            break;
        } // end switch on read mode

    // Check file mode
    switch (g_suI106Handle[iHandle].enFileState)
        {
        case enClosed :
            return I106_NOT_OPEN;
            break;

        case enWrite :
            return I106_WRONG_FILE_MODE;
            break;

        case enReadHeader :
        case enReadData   :
            // Backup to a point just before the most recently read header.
            // The amount to backup is the size of the previous header and the amount
            // of data already read.
            llInitialBackup = g_suI106Handle[iHandle].ulCurrHeaderBuffLen +
                              g_suI106Handle[iHandle].ulCurrDataBuffReadPos;
            break;

        case enReadUnsynced :
            llInitialBackup = 0;
            break;
        } // end switch file state


    // This puts us at the beginning of the most recently read header (or BOF)
    enI106Ch10GetPos(iHandle, &llCurrPos);
    llCurrPos = llCurrPos - llInitialBackup;

    // If at the beginning of the file then done, return BOF
    if (llCurrPos <= 0)
        {
        enI106Ch10SetPos(iHandle, 0);
        return I106_BOF;
        }

    // Loop until previous packet found
    bStillReading = bTRUE;
    while (bStillReading)
        {

        // Figure out how much to backup
        if (llCurrPos >= BACKUP_SIZE)
            iBackupAmount = BACKUP_SIZE;
        else
            iBackupAmount = (int)llCurrPos;

        // Backup that amount
        llNextBuffReadPos = llCurrPos - iBackupAmount;
        enI106Ch10SetPos(iHandle, llNextBuffReadPos);

        // Read a buffer of data to scan backwards through
        iReadCnt = read(g_suI106Handle[iHandle].iFile, abyScanBuff,
iBackupAmount+HEADER_SIZE);
```

```
        // Go to the end of the buffer and start scanning backwards
        for (iBuffIdx=iBackupAmount-1; iBuffIdx>=0; iBuffIdx--)
            {

            // Keep track of where we are in the file
            llCurrPos--;

            // Check for sync chars
            if ((abyScanBuff[iBuffIdx]   != 0x25) ||
                (abyScanBuff[iBuffIdx+1] != 0xEB))
                continue;

            // Sync chars found so check header checksum
            uCheckSum = uCalcHeaderChecksum((SuI106Ch10Header
*)(&abyScanBuff[iBuffIdx]));
            if (uCheckSum != ((SuI106Ch10Header *)(&abyScanBuff[iBuffIdx]))-
>uChecksum)
                continue;

            // Header checksum found so let ReadNextHeader() have a crack
            enStatus = enI106Ch10SetPos(iHandle, llCurrPos);
            if (enStatus != I106_OK)
                continue;
            enStatus = enI106Ch10ReadNextHeaderFile(iHandle, psuHeader);
            if (enStatus != I106_OK)
                continue;

            // Header OK so break out
            bStillReading = bFALSE;
            break;

            // At the beginning of the buffer go back and read some more

            } // end for all bytes in buffer

        // Check to see if we're at the BOF. BTW, if we're at the beginning of a file
        // and a valid header wasn't found then it IS a seek error.
        if (llCurrPos == 0)
            {
            bStillReading = bFALSE;
            enStatus = I106_SEEK_ERROR;
            }

        } // end while looping forever on buffers

    return enStatus;
    } // end enI106Ch10ReadPrevHeader()



/* ------------------------------------------------------------------------ */

// Get the next header. Depending on how the file was opened for reading,
// call the appropriate routine.

EnI106Status I106_CALL_DECL
    enI106Ch10ReadData(int                iHandle,
                       unsigned long      ulBuffSize,
                       void             * pvBuff)
    {
    EnI106Status    enStatus;
```

```
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_READ_NET_STREAM :
        case I106_READ :
        case I106_READ_IN_ORDER :
            enStatus = enI106Ch10ReadDataFile(iHandle, ulBuffSize, pvBuff);
            break;

        default :
            enStatus = I106_WRONG_FILE_MODE;
            break;

        } // end switch on read mode

    return enStatus;
    }



/* ------------------------------------------------------------------------ */

EnI106Status I106_CALL_DECL
    enI106Ch10ReadDataFile(int                iHandle,
                           unsigned long      ulBuffSize,
                           void             * pvBuff)
    {
    int             iReadCnt;
    unsigned long   ulReadAmount;

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Check for invalid file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
            return I106_WRONG_FILE_MODE;
            break;
        } // end switch on read mode

    // Check file state
    switch (g_suI106Handle[iHandle].enFileState)
        {
        case enClosed :
            return I106_NOT_OPEN;
            break;

        case enWrite :
            return I106_WRONG_FILE_MODE;
            break;

        case enReadData :
            break;
```

```
        default :
// MIGHT WANT TO SUPPORT THE "MORE DATA" METHOD INSTEAD
            g_suI106Handle[iHandle].enFileState = enReadUnsynced;
            return I106_READ_ERROR;
            break;
        } // end switch file state

    // Make sure there is enough room in the user buffer
// MIGHT WANT TO SUPPORT THE "MORE DATA" METHOD INSTEAD
    ulReadAmount = g_suI106Handle[iHandle].ulCurrDataBuffLen -
                   g_suI106Handle[iHandle].ulCurrDataBuffReadPos;
    if (ulBuffSize < ulReadAmount)
        return I106_BUFFER_TOO_SMALL;

    // Read the data, filler, and data checksum
    if (g_suI106Handle[iHandle].enFileMode != I106_READ_NET_STREAM)
        iReadCnt = read(g_suI106Handle[iHandle].iFile, pvBuff, ulReadAmount);
#if defined(IRIG_NETWORKING)
    else
        iReadCnt = enI106_ReadNetStream(iHandle, pvBuff, ulReadAmount);
#endif

    // If there was an error reading, figure out why
    if ((unsigned long)iReadCnt != ulReadAmount)
        {
        g_suI106Handle[iHandle].enFileState = enReadUnsynced;
        if (iReadCnt == -1)
            return I106_READ_ERROR;
        else
            return I106_EOF;
        } // end if read error

    // Keep track of our read position in the current data buffer
    g_suI106Handle[iHandle].ulCurrDataBuffReadPos = ulReadAmount;

// MAY WANT TO DO CHECKSUM CHECKING SOMEDAY

    // Expect a header next read
    g_suI106Handle[iHandle].enFileState = enReadHeader;

    return I106_OK;
    } // end enI106Ch10ReadData()



/* ------------------------------------------------------------------------ */

EnI106Status I106_CALL_DECL
    enI106Ch10WriteMsg(int                iHandle,
                       SuI106Ch10Header * psuHeader,
                       void             * pvBuff)
    {
    int     iHeaderLen;
    int     iWriteCnt;

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }
```

```
    // Check for invalid file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_READ            :
        case I106_READ_IN_ORDER   :
        case I106_READ_NET_STREAM :
            return I106_WRONG_FILE_MODE;
            break;
        } // end switch on read mode

    // Figure out header length
    iHeaderLen = iGetHeaderLen(psuHeader);

    // Write the header
    iWriteCnt = write(g_suI106Handle[iHandle].iFile, psuHeader, iHeaderLen);

    // If there was an error reading, figure out why
    if (iWriteCnt != iHeaderLen)
        {
        return I106_WRITE_ERROR;
        } // end if write error

    // Write the data
    iWriteCnt = write(g_suI106Handle[iHandle].iFile, pvBuff, psuHeader->ulPacketLen-
iHeaderLen);

    // If there was an error reading, figure out why
    if ((unsigned long)iWriteCnt != (psuHeader->ulPacketLen-iHeaderLen))
        {
        return I106_WRITE_ERROR;
        } // end if write error

    // Update the number of bytes written
    g_suI106Handle[iHandle].ulTotalBytesWritten += psuHeader->ulPacketLen;

    return I106_OK;
    }



/* ------------------------------------------------------------------------
 * Move file pointer
 * ------------------------------------------------------------------------ */

EnI106Status I106_CALL_DECL
    enI106Ch10FirstMsg(int iHandle)
    {

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Check file modes
    switch (g_suI106Handle[iHandle].enFileMode)
```

```
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
        case I106_READ_NET_STREAM :
        default                   :
            return I106_WRONG_FILE_MODE;
            break;

        case I106_READ_IN_ORDER   :
            g_suI106Handle[iHandle].suInOrderIndex.iArrayCurr = 0;
            enI106Ch10SetPos(iHandle, 0L);
            break;

        case I106_READ :
            enI106Ch10SetPos(iHandle, 0L);
            break;
        } // end switch on read mode

    return I106_OK;
    }



/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10LastMsg(int iHandle)
    {
    EnI106Status        enReturnStatus;
    EnI106Status        enStatus;
    int64_t             llPos;
    SuI106Ch10Header    suHeader;
    int                 iReadCnt;
#if !defined(_MSC_VER)
    struct stat         suStatBuff;
#endif

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }


    // Check file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
        case I106_READ_NET_STREAM :
        default                   :
            return I106_WRONG_FILE_MODE;
            break;
```

```
        // If its opened for reading in order then just set the index pointer
        // to the last index.
        case I106_READ_IN_ORDER   :
            g_suI106Handle[iHandle].suInOrderIndex.iArrayCurr =
g_suI106Handle[iHandle].suInOrderIndex.iArrayUsed-1;
            enReturnStatus = I106_OK;
            break;

        // If there is no index then do it the hard way
        case I106_READ :

            // Figure out how big the file is and go to the end
#if defined(_MSC_VER)
            llPos = _filelengthi64(g_suI106Handle[iHandle].iFile) - HEADER_SIZE;
#else
            fstat(g_suI106Handle[iHandle].iFile, &suStatBuff);
            llPos = suStatBuff.st_size - HEADER_SIZE;
#endif

            //if ((llPos % 4) != 0)
            //    return I106_SEEK_ERROR;

            // Now loop forever looking for a valid packet or die trying
            while (1==1)
                {
                // Not at the beginning so go back 1 byte and try again
                llPos -= 1;

                // Go to the new position and look for a legal header
                enStatus = enI106Ch10SetPos(iHandle, llPos);
                if (enStatus != I106_OK)
                    return I106_SEEK_ERROR;

                // Read and check the header
                iReadCnt = read(g_suI106Handle[iHandle].iFile, &suHeader,
HEADER_SIZE);

                if (iReadCnt != HEADER_SIZE)
                    continue;

                if (suHeader.uSync != IRIG106_SYNC)
                    continue;

                // Sync pattern matched so check the header checksum
                if (suHeader.uChecksum == uCalcHeaderChecksum(&suHeader))
                    {
                    enReturnStatus = I106_OK;
                    break;
                    }

                // No match, check for begining of file
    // ONLY NEED TO GO BACK THE MAX PACKET SIZE
                if (llPos <= 0)
                    {
                    enReturnStatus = I106_SEEK_ERROR;
                    break;
                    }

                } // end looping forever

            // Go back to the good position
            enStatus = enI106Ch10SetPos(iHandle, llPos);
```

```
            break;
        } // end switch on read mode

    return enReturnStatus;
    }


/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10SetPos(int iHandle, int64_t llOffset)
    {

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }


    // Check file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_OVERWRITE       :
        case I106_APPEND          :
        case I106_READ_NET_STREAM :
        default                   :
            return I106_WRONG_FILE_MODE;
            break;

        case I106_READ_IN_ORDER   :
        case I106_READ :
            // Seek
#if defined(_WIN32)
            {
            __int64  llStatus;
            llStatus = _lseeki64(g_suI106Handle[iHandle].iFile, llOffset, SEEK_SET);
            }
#else
    {
    off64_t  llStatus;
    llStatus = lseek64(g_suI106Handle[iHandle].iFile, (off64_t)llOffset, SEEK_SET);
    assert(llStatus >= 0);
    }
#endif

            // Can't be sure we're on a message boundary so set unsync'ed
            g_suI106Handle[iHandle].enFileState = enReadUnsynced;
            break;
        } // end switch on file mode

    return I106_OK;
    }
```

```
/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10GetPos(int iHandle, int64_t *pllOffset)
    {

    // Check for a valid handle
    if ((iHandle <  0)           ||
        (iHandle >= MAX_HANDLES) ||
        (g_suI106Handle[iHandle].bInUse == bFALSE))
        {
        return I106_INVALID_HANDLE;
        }

    // Check file modes
    switch (g_suI106Handle[iHandle].enFileMode)
        {
        case I106_CLOSED :
            return I106_NOT_OPEN;
            break;

        case I106_READ_NET_STREAM :
        default                   :
            return I106_WRONG_FILE_MODE;
            break;

        case I106_READ_IN_ORDER   :
        case I106_READ            :
        case I106_OVERWRITE       :
        case I106_APPEND          :
    // Get position
#if defined(_WIN32)
            *pllOffset = _telli64(g_suI106Handle[iHandle].iFile);
#else
    {
    *pllOffset = (int64_t)lseek64(g_suI106Handle[iHandle].iFile, (off64_t)0,
SEEK_CUR);
    assert(*pllOffset >= 0);
    }
#endif
            break;
        } // end switch on file mode


    return I106_OK;
    }



/* ----------------------------------------------------------------------
 * Utilities
 * ---------------------------------------------------------------------- */

/* Set packet header to some sane values. Be sure to fill in proper values for:
        ulPacketLen
        ulDataLen
        ubyHdrVer
        aubyRefTime
        uChecksum
 */
int I106_CALL_DECL
    iHeaderInit(SuI106Ch10Header * psuHeader,
```

```
                unsigned int       uChanID,
                unsigned int       uDataType,
                unsigned int       uFlags,
                unsigned int       uSeqNum)
    {

    // Make a legal, valid header
    psuHeader->uSync          = IRIG106_SYNC;
    psuHeader->uChID          = uChanID;
    psuHeader->ulPacketLen    = HEADER_SIZE;
    psuHeader->ulDataLen      = 0;
    psuHeader->ubyHdrVer      = 0x02;  // <-- NEED TO PASS THIS IN!!!
    psuHeader->ubySeqNum      = uSeqNum;
    psuHeader->ubyPacketFlags = uFlags;
    psuHeader->ubyDataType    = uDataType;
    memset(&(psuHeader->aubyRefTime), 0, 6);
//  psuHeader->uChecksum      = uCalcHeaderChecksum(psuHeader);
    psuHeader->uChecksum      = 0;
// This could overrun if the header doesn't have a secondary header
//    memset(&(psuHeader->aulTime), 0, 8);
//    psuHeader->uReserved      = 0;
//    psuHeader->uSecChecksum   = uCalcSecHeaderChecksum(psuHeader);

    return 0;
    }

/* ---------------------------------------------------------------------- */

// Figure out header length (might need to check header version at
// some point if I can ever figure out what the different header
// version mean.

int I106_CALL_DECL
    iGetHeaderLen(SuI106Ch10Header * psuHeader)
    {
    int     iHeaderLen;

    if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_SEC_HEADER) == 0)
        iHeaderLen = HEADER_SIZE;
    else
        iHeaderLen = HEADER_SIZE + SEC_HEADER_SIZE;

    return iHeaderLen;
    }




/* ---------------------------------------------------------------------- */

// Figure out data length including padding and any data checksum

uint32_t I106_CALL_DECL
    uGetDataLen(SuI106Ch10Header * psuHeader)
    {
    int     iDataLen;

    iDataLen = psuHeader->ulPacketLen - iGetHeaderLen(psuHeader);

    return iDataLen;
    }
```

```
/* ------------------------------------------------------------------------ */

uint16_t I106_CALL_DECL
    uCalcHeaderChecksum(SuI106Ch10Header * psuHeader)
    {
    int             iHdrIdx;
    uint16_t        uHdrSum;
    uint16_t      * aHdr = (uint16_t *)psuHeader;

    uHdrSum = 0;
    for (iHdrIdx=0; iHdrIdx<(HEADER_SIZE-2)/2; iHdrIdx++)
        uHdrSum += aHdr[iHdrIdx];

    return uHdrSum;
    }


/* ------------------------------------------------------------------------ */

uint16_t I106_CALL_DECL
    uCalcSecHeaderChecksum(SuI106Ch10Header * psuHeader)
    {

    int             iByteIdx;
    uint16_t        uHdrSum;
// MAKE THIS 16 BIT UNSIGNEDS LIKE ABOVE
    unsigned char * auchHdrByte = (unsigned char *)psuHeader;

    uHdrSum = 0;
    for (iByteIdx=0; iByteIdx<SEC_HEADER_SIZE-2; iByteIdx++)
        uHdrSum += auchHdrByte[iByteIdx+HEADER_SIZE];

    return uHdrSum;
    }


/* ------------------------------------------------------------------------ */

// Calculate and return the required size of the data buffer portion of the
// packet including checksum and appropriate filler for 4 byte alignment.

uint32_t I106_CALL_DECL
    uCalcDataBuffReqSize(uint32_t uDataLen, int iChecksumType)
    {
    uint32_t    uDataBuffLen;

    // Start with the length of the data
    uDataBuffLen = uDataLen;

    // Add in enough for the selected checksum
    switch (iChecksumType)
        {
        case I106CH10_PFLAGS_CHKSUM_NONE :
            break;
        case I106CH10_PFLAGS_CHKSUM_8    :
            uDataBuffLen += 1;
            break;
        case I106CH10_PFLAGS_CHKSUM_16   :
            uDataBuffLen += 2;
            break;
        case I106CH10_PFLAGS_CHKSUM_32   :
            uDataBuffLen += 4;
```

```
            break;
        default :
            uDataBuffLen = 0;
        } // end switch iChecksumType

    // Now add filler for 4 byte alignment
    uDataBuffLen += 3;
    uDataBuffLen &= 0xfffffffc;

    return uDataBuffLen;
    }


/* ---------------------------------------------------------------------- */

// Add the filler and appropriate checksum to the end of the data buffer
// It is assumed that the buffer is big enough to hold additional filler
// and the checksum. Also fill in the header with the correct packet length.

EnI106Status I106_CALL_DECL
    uAddDataFillerChecksum(SuI106Ch10Header * psuI106Hdr, unsigned char achData[])
    {
    uint32_t    uDataIdx;
    uint32_t    uDataBuffSize;
    uint32_t    uFillSize;
    int         iChecksumType;

    uint8_t    *puSum8;
    uint8_t    *puData8;
    uint16_t   *puSum16;
    uint16_t   *puData16;
    uint32_t   *puSum32;
    uint32_t   *puData32;

    // Extract the checksum type
    iChecksumType = psuI106Hdr->ubyPacketFlags & 0x03;

    // Figure out how big the final packet will be
    uDataBuffSize = uCalcDataBuffReqSize(psuI106Hdr->ulDataLen, iChecksumType);
    psuI106Hdr->ulPacketLen = HEADER_SIZE + uDataBuffSize;
    if ((psuI106Hdr->ubyPacketFlags & I106CH10_PFLAGS_SEC_HEADER) != 0)
        psuI106Hdr->ulPacketLen += SEC_HEADER_SIZE;

    // Figure out the filler/checksum size and zero fill it
    uFillSize = uDataBuffSize - psuI106Hdr->ulDataLen;
    memset(&achData[psuI106Hdr->ulDataLen], 0, uFillSize);

    // If no checksum then we're done
    if (iChecksumType == I106CH10_PFLAGS_CHKSUM_NONE)
        return I106_OK;

    // Calculate the checksum
    switch (iChecksumType)
        {
        case I106CH10_PFLAGS_CHKSUM_8    :
            // Checksum the data and filler
            puData8 = (uint8_t *)achData;
            puSum8  = (uint8_t *)&achData[psuI106Hdr->ulDataLen+uFillSize-1];
            for (uDataIdx=0; uDataIdx<uDataBuffSize-1; uDataIdx++)
                {
                *puSum8 += *puData8;
                puData8++;
                }
```

```
            break;

        case I106CH10_PFLAGS_CHKSUM_16   :
            puData16 = (uint16_t *)achData;
            puSum16  = (uint16_t *)&achData[psuI106Hdr->ulDataLen+uFillSize-2];
            for (uDataIdx=0; uDataIdx<(uDataBuffSize/2)-1; uDataIdx++)
                {
                *puSum16 += *puData16;
                puData16++;
                }
            break;

        case I106CH10_PFLAGS_CHKSUM_32   :
            puData32 = (uint32_t *)achData;
            puSum32  = (uint32_t *)&achData[psuI106Hdr->ulDataLen+uFillSize-4];
            for (uDataIdx=0; uDataIdx<(uDataBuffSize/4)-1; uDataIdx++)
                {
                *puSum32 += *puData32;
                puData32++;
                }
            break;
        default :
            break;
        } // end switch iChecksumType

    return I106_OK;
    }


// ----------------------------------------------------------------------

char * szI106ErrorStr(EnI106Status enStatus)
    {
    char    * szErrorMsg;

    switch (enStatus)
        {
        case I106_OK                : szErrorMsg = "No error";              break;
        case I106_OPEN_ERROR        : szErrorMsg = "File open failed";      break;
        case I106_OPEN_WARNING      : szErrorMsg = "File open warning";     break;
        case I106_EOF               : szErrorMsg = "End of file";           break;
        case I106_BOF               : szErrorMsg = "Beginning of file";     break;
        case I106_READ_ERROR        : szErrorMsg = "Read error";            break;
        case I106_WRITE_ERROR       : szErrorMsg = "Write error";           break;
        case I106_MORE_DATA         : szErrorMsg = "More data available";   break;
        case I106_SEEK_ERROR        : szErrorMsg = "Seek error";            break;
        case I106_WRONG_FILE_MODE   : szErrorMsg = "Wrong file mode";       break;
        case I106_NOT_OPEN          : szErrorMsg = "File not open";         break;
        case I106_ALREADY_OPEN      : szErrorMsg = "File already open";     break;
        case I106_BUFFER_TOO_SMALL  : szErrorMsg = "Buffer too small";      break;
        case I106_NO_MORE_DATA      : szErrorMsg = "No more data";          break;
        case I106_NO_FREE_HANDLES   : szErrorMsg = "No free file handles";  break;
        case I106_INVALID_HANDLE    : szErrorMsg = "Invalid file handle";   break;
        case I106_TIME_NOT_FOUND    : szErrorMsg = "Time not found";        break;
        case I106_HEADER_CHKSUM_BAD : szErrorMsg = "Bad header checksum";   break;
        case I106_NO_INDEX          : szErrorMsg = "No index";              break;
        case I106_UNSUPPORTED       : szErrorMsg = "Unsupported feature";   break;
        case I106_BUFFER_OVERRUN    : szErrorMsg = "Buffer overrun";        break;
        case I106_INDEX_NODE        : szErrorMsg = "Index node";            break;
        case I106_INDEX_ROOT        : szErrorMsg = "Index root";            break;
        case I106_INDEX_ROOT_LINK   : szErrorMsg = "Index root link";       break;
        case I106_INVALID_DATA      : szErrorMsg = "Invalid data";          break;
        case I106_INVALID_PARAMETER : szErrorMsg = "Invalid parameter";     break;
```

```
        default                     : szErrorMsg = "Unknown error";         break;
        } // end switch on status

    return szErrorMsg;
    }


// ---------------------------------------------------------------------

void InitHandles()
    {
    int     iIdx;

    // Initialize handle data if necessary
    if (m_bHandlesInited == bFALSE)
        {
        for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
            {
            g_suI106Handle[iIdx].bInUse      = bFALSE;
            g_suI106Handle[iIdx].enFileMode  = I106_CLOSED;
            g_suI106Handle[iIdx].enFileState = enClosed;
            }
        m_bHandlesInited = bTRUE;
        } // end if file handles not inited yet
    }


// ---------------------------------------------------------------------

// Get the next available handle

int GetNextHandle()
    {
    int     iHandle;
    int     iIdx;

    iHandle = -1;
    for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
        {
        if (g_suI106Handle[iIdx].bInUse == bFALSE)
            {
            g_suI106Handle[iIdx].bInUse = bTRUE;
            iHandle = iIdx;
            break;
            } // end if unused handle found
        } // end looking for unused handle

    return iHandle;
    }

#ifdef __cplusplus
} // end namespace
#endif
```

Appendix A-3. i106_time.h

```
/*****************************************************************************

 i106_time.h -

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 *****************************************************************************/

#ifndef _I106_TIME_H
#define _I106_TIME_H

//#include "irig106ch10.h"
#include <time.h>

#ifdef __cplusplus
namespace Irig106 {
extern "C" {
#endif

/*
 * Macros and definitions
 * ----------------------
 */

#define CH4BINARYTIME_HIGH_LSB_SEC   655.36
#define CH4BINARYTIME_LOW_LSB_SEC      0.01
#define _100_NANO_SEC_IN_MICRO_SEC    10

typedef PUBLIC enum DateFmt
    {
    I106_DATEFMT_DAY         =  0,
    I106_DATEFMT_DMY         =  1,
    } EnI106DateFmt;
```

```
/*
 * Data structures
 * ---------------
 */

// Time has a number of representations in the IRIG 106 spec.
// The structure below is used as a convenient standard way of
// representing time. The nice thing about standards is that there
// are so many to choose from, and time is no exception. But none of
// the various C time representations really fill the bill. So I made
// a new time representation. So there.
typedef PUBLIC struct SuIrig106Time_S
    {
//  uint32_t        ulSecs;     // This is a time_t
    time_t          ulSecs;
    uint32_t        ulFrac;     // LSB = 100ns
    EnI106DateFmt   enFmt;      // Day or DMY format
    } SuIrig106Time;


// Relative time to absolute time reference
typedef PUBLIC struct SuTimeRef_S
    {
    int64_t         uRelTime;          // Relative time from header
    SuIrig106Time   suIrigTime;        // Clock time from IRIG source
    uint16_t        bRelTimeValid   :  1;
    uint16_t        bAbsTimeValid   :  1;
    uint16_t        uReserved       :  14;
    } SuTimeRef;


/// IRIG 106 secondary header time in Ch 4 BCD format
typedef PUBLIC struct SuI106Ch4_BCD_Time_S
    {
    uint16_t      uMin1     : 4;    ///< High order time
    uint16_t      uMin10    : 3;
    uint16_t      uHour1    : 4;
    uint16_t      uHour10   : 2;
    uint16_t      uDay1     : 3;
    uint16_t      uSec0_01  : 4;    ///< Low order time
    uint16_t      uSec0_1   : 4;
    uint16_t      uSec1     : 4;
    uint16_t      uSec10    : 2;
    uint16_t      uReserved : 2;
    uint16_t      uUSecs;           ///< Microsecond time
#if !defined(__GNUC__)
    } SuI106Ch4_BCD_Time;
#else
    } __attribute__ ((packed)) SuI106Ch4_BCD_Time;
#endif


/// IRIG 106 secondary header time in Ch 4 binary format
typedef PUBLIC struct SuI106Ch4_Binary_Time_S
    {
    uint16_t      uHighBinTime;     ///< High order time
    uint16_t      uLowBinTime;      ///< Low order time
    uint16_t      uUSecs;           ///< Microsecond time
#if !defined(__GNUC__)
    } SuI106Ch4_Binary_Time;
#else
```

```
    } __attribute__ ((packed)) SuI106Ch4_Binary_Time;
#endif


/// IRIG 106 secondary header time in IEEE-1588 format
typedef PUBLIC struct SuIEEE1588_Time_S
    {
    uint32_t      uNanoSeconds;     ///< Nano-seconds
    uint32_t      uSeconds;         ///< Seconds
#if !defined(__GNUC__)
    } SuIEEE1588_Time;
#else
    } __attribute__ ((packed)) SuIEEE1588_Time;
#endif

/// IRIG 106 secondary header time in Extended RTC format
typedef PUBLIC struct SuERTC_Time_S
    {
    uint32_t      uLSLW;            ///< Least significant long word
    uint32_t      uMSLW;            ///< Most significant long word
#if !defined(__GNUC__)
    } SuERTC_Time;
#else
    } __attribute__ ((packed)) SuERTC_Time;
#endif


/// Intra-packet header relative time counter format
typedef PUBLIC struct SuIntraPacketRtc_S
    {
    uint8_t       aubyRefTime[6];   // Reference time
    uint16_t      uReserved;
#if !defined(__GNUC__)
    } SuIntraPacketRtc;
#else
    } __attribute__ ((packed)) SuIntraPacketRtc;
#endif

/// Intra-packet header time stamp - raw data
typedef PUBLIC struct SuIntraPacketTS_S
    {
    uint8_t       aubyIntPktTime[8];   // Time Stamp
#if !defined(__GNUC__)
    } SuIntraPacketTS;
#else
    } __attribute__ ((packed)) SuIntraPacketTS;
#endif

/*
 * Global data
 * -----------
 */


/*
 * Function Declaration
 * --------------------
 */

//EnI106Status I106_CALL_DECL
//    enI106_SetRelTime(int             iI106Ch10Handle,
//                      SuIrig106Time  * psuTime,
//                      uint8_t          abyRelTime[]);
```

```
EnI106Status I106_CALL_DECL
    enI106_SetRelTime(int              iI106Ch10Handle,
                      SuIrig106Time  * psuTime,
                      uint8_t          abyRelTime[]);

EnI106Status I106_CALL_DECL
    enI106_Rel2IrigTime(int                 iI106Ch10Handle,
                        uint8_t             abyRelTime[],
                        SuIrig106Time     * psuTime);

EnI106Status I106_CALL_DECL
    enI106_RelInt2IrigTime(int              iI106Ch10Handle,
                           int64_t          llRelTime,
                           SuIrig106Time  * psuTime);

EnI106Status I106_CALL_DECL
    enI106_Irig2RelTime(int             iI106Ch10Handle,
                        SuIrig106Time  * psuTime,
                        uint8_t          abyRelTime[]);

EnI106Status I106_CALL_DECL
    enI106_Ch4Binary2IrigTime(SuI106Ch4_Binary_Time * psuCh4BinaryTime,
                              SuIrig106Time         * psuIrig106Time);

EnI106Status I106_CALL_DECL
    enI106_IEEE15882IrigTime(SuIEEE1588_Time * psuIEEE1588Time,
                             SuIrig106Time   * psuIrig106Time);

EnI106Status I106_CALL_DECL
    vFillInTimeStruct(SuI106Ch10Header  * psuHeader,
                      SuIntraPacketTS   * psuIntraPacketTS,
                      SuTimeRef         * psuTimeRef);

// Warning - array to int / int to array functions are little endian only!

void I106_CALL_DECL
    vLLInt2TimeArray(int64_t * pllRelTime,
                     uint8_t   abyRelTime[]);


void I106_CALL_DECL
    vTimeArray2LLInt(uint8_t   abyRelTime[],
                     int64_t * pllRelTime);


EnI106Status I106_CALL_DECL
    enI106_SyncTime(int     iI106Ch10Handle,
                    int     bRequireSync,       // Require external time sync
                    int     iTimeLimit);        // Max scan ahead time in seconds, 0 =
no limit

EnI106Status I106_CALL_DECL
    enI106Ch10SetPosToIrigTime(int iI106Ch10Handle, SuIrig106Time * psuSeekTime);


// General purpose time utilities
// ------------------------------

// Convert IRIG time into an appropriate string
char * IrigTime2String(SuIrig106Time * psuTime);

// This is handy enough that we'll go ahead and export it to the world
uint32_t I106_CALL_DECL mkgmtime(struct tm * psuTmTime);
```

```
#ifdef __cplusplus
} // end extern "C"
} // end namespace i106
#endif

#endif
```

Appendix A-4. i106_time.c

```
/****************************************************************************

 i106_time.c -

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>

#include "stdint.h"
#include "irig106ch10.h"
#include "i106_time.h"
#include "i106_decode_time.h"

#ifdef __cplusplus
namespace Irig106 {
#endif

/*
 * Macros and definitions
 * ----------------------
 */


/*
 * Data structures
 * ---------------
 */
```

```
/*
 * Module data
 * -----------
 */

static SuTimeRef    m_asuTimeRef[MAX_HANDLES]; // Relative / absolute time reference

/*
 * Function Declaration
 * --------------------
 */


/* ------------------------------------------------------------------------ */

// Update the current reference time value
EnI106Status I106_CALL_DECL
    enI106_SetRelTime(int              iI106Ch10Handle,
                      SuIrig106Time  * psuTime,
                      uint8_t          abyRelTime[])
    {

    // Save the absolute time value
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs = psuTime->ulSecs;
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac = psuTime->ulFrac;
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.enFmt  = psuTime->enFmt;

    // Save the relative (i.e. the 10MHz counter) value
    m_asuTimeRef[iI106Ch10Handle].uRelTime          = 0;
    memcpy((char *)&(m_asuTimeRef[iI106Ch10Handle].uRelTime),
           (char *)&abyRelTime[0], 6);

    return I106_OK;
    }



/* ------------------------------------------------------------------------ */

// Take a 6 byte relative time value (like the one in the IRIG header) and
// turn it into a real time based on the current reference IRIG time.

EnI106Status I106_CALL_DECL
    enI106_Rel2IrigTime(int              iI106Ch10Handle,
                        uint8_t          abyRelTime[],
                        SuIrig106Time  * psuTime)
    {
    int64_t         llRelTime;
    EnI106Status    enStatus;

    // Convert 6 byte time array to 16 bit int. This only works for
    // positive time, but that shouldn't be a problem
    llRelTime = 0L;
    memcpy(&llRelTime, &abyRelTime[0], 6);

    enStatus = enI106_RelInt2IrigTime(iI106Ch10Handle, llRelTime, psuTime);

    return enStatus;
    }
```

```
/* ---------------------------------------------------------------------- */

// Take a 64 bit relative time value and turn it into a real time based on
// the current reference IRIG time.

EnI106Status I106_CALL_DECL
    enI106_RelInt2IrigTime(int               iI106Ch10Handle,
                           int64_t           llRelTime,
                           SuIrig106Time   * psuTime)
    {
    int64_t         uTimeDiff;
    int64_t         lFracDiff;
    int64_t         lSecDiff;

    int64_t         lSec;
    int64_t         lFrac;


    // Figure out the relative time difference
    uTimeDiff = llRelTime - m_asuTimeRef[iI106Ch10Handle].uRelTime;
    lSecDiff  = uTimeDiff / 10000000;
    lFracDiff = uTimeDiff % 10000000;

    lSec      = m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs + lSecDiff;
    lFrac     = m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac + lFracDiff;

    // This seems a bit extreme but it's defensive programming
    while (lFrac < 0)
        {
        lFrac += 10000000;
        lSec  -= 1;
        }

    while (lFrac >= 10000000)
        {
        lFrac -= 10000000;
        lSec  += 1;
        }

    // Now add the time difference to the last IRIG time reference
    psuTime->ulFrac = (unsigned long)lFrac;
    psuTime->ulSecs = (unsigned long)lSec;
    psuTime->enFmt  = m_asuTimeRef[iI106Ch10Handle].suIrigTime.enFmt;

    return I106_OK;
    }



/* ---------------------------------------------------------------------- */

// Take a real clock time and turn it into a 6 byte relative time.

EnI106Status I106_CALL_DECL
    enI106_Irig2RelTime(int              iI106Ch10Handle,
                        SuIrig106Time  * psuTime,
                        uint8_t          abyRelTime[])
    {
    int64_t         llDiff;
    int64_t         llNewRel;

    // Calculate time difference (LSB = 100 nSec) between the passed time
```

```
    // and the time reference
    llDiff =
         (int64_t)(+ psuTime->ulSecs -
m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs) * 10000000 +
         (int64_t)(+ psuTime->ulFrac -
m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac);

    // Add this amount to the reference
    llNewRel = m_asuTimeRef[iI106Ch10Handle].uRelTime + llDiff;

    // Now convert this to a 6 byte relative time
    memcpy((char *)&abyRelTime[0],
           (char *)&(llNewRel), 6);

    return I106_OK;
    }

/* ---------------------------------------------------------------------- */

// Take a Irig Ch4 time value (like the one in a secondary IRIG header) and
// turn it into an Irig106 time

EnI106Status I106_CALL_DECL
    enI106_Ch4Binary2IrigTime(SuI106Ch4_Binary_Time * psuCh4BinaryTime,
                              SuIrig106Time         * psuIrig106Time)
{
    psuIrig106Time->ulSecs = (unsigned long)
        ( (double)psuCh4BinaryTime->uHighBinTime * CH4BINARYTIME_HIGH_LSB_SEC
        + (unsigned long)psuCh4BinaryTime->uLowBinTime * CH4BINARYTIME_LOW_LSB_SEC );
    psuIrig106Time->ulFrac = (unsigned long)psuCh4BinaryTime->uUSecs *
_100_NANO_SEC_IN_MICRO_SEC;

    return I106_OK;
}

/* ---------------------------------------------------------------------- */

// Take a IEEE-1588 time value (like the one in a secondary IRIG header) and
// turn it into an Irig106 time
EnI106Status I106_CALL_DECL
    enI106_IEEE15882IrigTime(SuIEEE1588_Time * psuIEEE1588Time,
                              SuIrig106Time  * psuIrig106Time)
{
    psuIrig106Time->ulSecs = (unsigned long)psuIEEE1588Time->uSeconds;
    //Convert 'nanoseconds' to '100 nanoseconds'
    psuIrig106Time->ulFrac = (unsigned long)psuIEEE1588Time->uNanoSeconds/100;

    return I106_OK;
}


/* ---------------------------------------------------------------------- */

// Warning - array to int / int to array functions are little endian only!

// Create a 6 byte array value from a 64 bit int relative time

void I106_CALL_DECL
    vLLInt2TimeArray(int64_t * pllRelTime,
                     uint8_t   abyRelTime[])
    {
    memcpy((char *)abyRelTime, (char *)pllRelTime, 6);
    return;
```

```
    }


/* ---------------------------------------------------------------------- */

// Create a 64 bit int relative time from 6 byte array value

void I106_CALL_DECL
    vTimeArray2LLInt(uint8_t   abyRelTime[],
                     int64_t * pllRelTime)
    {
    *pllRelTime = 0L;
    memcpy((char *)pllRelTime, (char *)abyRelTime, 6);
    return;
    }



/* ---------------------------------------------------------------------- */

// Read the data file from the current position to try to determine a valid
// relative time to clock time from a time packet.

EnI106Status I106_CALL_DECL
    enI106_SyncTime(int     iI106Ch10Handle,
                    int     bRequireSync,   // Require external time source sync
                    int     iTimeLimit)     // Max time to look in seconds
    {
    int64_t             llCurrOffset;
    int64_t             llTimeLimit;
    int64_t             llCurrTime;
    EnI106Status        enStatus;
    EnI106Status        enRetStatus;
    SuI106Ch10Header    suI106Hdr;
    SuIrig106Time       suTime;
    unsigned long       ulBuffSize = 0;
    void              * pvBuff = NULL;
    SuTimeF1_ChanSpec * psuChanSpecTime = NULL;

    // Get and save the current file position
    enStatus = enI106Ch10GetPos(iI106Ch10Handle, &llCurrOffset);
    if (enStatus != I106_OK)
        return enStatus;

    // Set the file to the start
    //enStatus = enI106Ch10SetPos(iI106Ch10Handle, 0);
    //if (enStatus != I106_OK)
    //    return enStatus;

    // Read the next header
    enStatus = enI106Ch10ReadNextHeaderFile(iI106Ch10Handle, &suI106Hdr);
    if (enStatus == I106_EOF)
        return I106_TIME_NOT_FOUND;

    if (enStatus != I106_OK)
        return enStatus;

    // Calculate the time limit if there is one
    if (iTimeLimit > 0)
        {
        vTimeArray2LLInt(suI106Hdr.aubyRefTime, &llTimeLimit);
        llTimeLimit = llTimeLimit + (int64_t)iTimeLimit * (int64_t)10000000;
        }
```

```
else
    llTimeLimit = 0;

// Loop, looking for appropriate time message
while (bTRUE)
    {

    // See if we've passed our time limit
    if (llTimeLimit > 0)
        {
        vTimeArray2LLInt(suI106Hdr.aubyRefTime, &llCurrTime);
        if (llTimeLimit < llCurrTime)
            {
            enRetStatus = I106_TIME_NOT_FOUND;
            break;
            }
        } // end if there is a time limit

    // If IRIG time type then process it
    if (suI106Hdr.ubyDataType == I106CH10_DTYPE_IRIG_TIME)
        {

        // Read header OK, make buffer for time message
        if (ulBuffSize < suI106Hdr.ulPacketLen)
            {
            pvBuff          = realloc(pvBuff, suI106Hdr.ulPacketLen);
            psuChanSpecTime = (SuTimeF1_ChanSpec *)pvBuff;
            ulBuffSize      = suI106Hdr.ulPacketLen;
            }

        // Read the data buffer
        enStatus = enI106Ch10ReadData(iI106Ch10Handle, ulBuffSize, pvBuff);
        if (enStatus != I106_OK)
            {
            enRetStatus = I106_TIME_NOT_FOUND;
            break;
            }

        // If external sync OK then decode it and set relative time
        if ((bRequireSync == bFALSE) || (psuChanSpecTime->uTimeSrc == 1))
            {
            enI106_Decode_TimeF1(&suI106Hdr, pvBuff, &suTime);
            enI106_SetRelTime(iI106Ch10Handle, &suTime, suI106Hdr.aubyRefTime);
            enRetStatus = I106_OK;
            break;
            }
        } // end if IRIG time message

    // Read the next header and try again
    enStatus = enI106Ch10ReadNextHeaderFile(iI106Ch10Handle, &suI106Hdr);
    if (enStatus == I106_EOF)
        {
        enRetStatus = I106_TIME_NOT_FOUND;
        break;
        }

    if (enStatus != I106_OK)
        {
        enRetStatus = enStatus;
        break;
        }

    } // end while looping looking for time message
```

```
    // Restore file position
    enStatus = enI106Ch10SetPos(iI106Ch10Handle, llCurrOffset);
    if (enStatus != I106_OK)
        {
        enRetStatus = enStatus;
        }

    // Return the malloc'ed memory
    free(pvBuff);

    return enRetStatus;
    }


/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10SetPosToIrigTime(int iHandle, SuIrig106Time * psuSeekTime)
    {
    uint8_t             abySeekTime[6];
    int64_t             llSeekTime;
    SuInOrderIndex    * psuIndex = &g_suI106Handle[iHandle].suInOrderIndex;
    int                 iUpperLimit;
    int                 iLowerLimit;
    int                 iSearchLoopIdx;

    // If there is no index in memory then barf
    if (psuIndex->enSortStatus != enSorted)
        return I106_NO_INDEX;

    // We have an index so do a binary search for time

    // Convert clock time to 10 MHz count
    enI106_Irig2RelTime(iHandle, psuSeekTime, abySeekTime);
    vTimeArray2LLInt(abySeekTime, &llSeekTime);

    // Check time bounds
    if (llSeekTime < psuIndex->asuIndex[0].llTime)
        {
        enI106Ch10FirstMsg(iHandle);
        return I106_TIME_NOT_FOUND;
        };

    if (llSeekTime > psuIndex->asuIndex[psuIndex->iArrayUsed].llTime)
        {
        enI106Ch10LastMsg(iHandle);
        return I106_TIME_NOT_FOUND;
        };

    // If we don't already have it, figure out how many search steps
    if (psuIndex->iNumSearchSteps == 0)
        {
        iUpperLimit = 1;
        while (iUpperLimit < psuIndex->iArrayUsed)
            {
            iUpperLimit *= 2;
            psuIndex->iNumSearchSteps++;
            }
        } // end if no search steps

    // Loop prescribed number of times
    iLowerLimit = 0;
```

```
    iUpperLimit = psuIndex->iArrayUsed-1;
    psuIndex->iArrayCurr = (iUpperLimit - iLowerLimit) / 2;
    for (iSearchLoopIdx = 0;
         iSearchLoopIdx < psuIndex->iNumSearchSteps;
         iSearchLoopIdx++)
        {
        if      (psuIndex->asuIndex[psuIndex->iArrayCurr].llTime > llSeekTime)
            iUpperLimit = (iUpperLimit - iLowerLimit) / 2;
        else if (psuIndex->asuIndex[psuIndex->iArrayCurr].llTime < llSeekTime)
            iLowerLimit = (iUpperLimit - iLowerLimit) / 2;
        else
            break;
        }

    return I106_OK;
    }


/* ---------------------------------------------------------------------- */

// General purpose time utilities
// ------------------------------

// Convert IRIG time into an appropriate string

char * IrigTime2String(SuIrig106Time * psuTime)
    {
    static char     szTime[30];
    struct tm     * psuTmTime;

    // Convert IRIG time into it's components
    psuTmTime = gmtime((time_t *)&(psuTime->ulSecs));

    // Make the appropriate string
    switch (psuTime->enFmt)
        {
        // Year / Month / Day format ("2008/02/29 12:34:56.789")
        case I106_DATEFMT_DMY :
            sprintf(szTime, "%4.4i/%2.2i/%2.2i %2.2i:%2.2i:%2.2i.%3.3i",
                psuTmTime->tm_year + 1900,
                psuTmTime->tm_mon + 1,
                psuTmTime->tm_mday,
                psuTmTime->tm_hour,
                psuTmTime->tm_min,
                psuTmTime->tm_sec,
                psuTime->ulFrac / 10000);
            break;

        // Day of the Year format ("001:12:34:56.789")
        case I106_DATEFMT_DAY :
        default :
            sprintf(szTime, "%3.3i:%2.2i:%2.2i:%2.2i.%3.3i",
                psuTmTime->tm_yday+1,
                psuTmTime->tm_hour,
                psuTmTime->tm_min,
                psuTmTime->tm_sec,
                psuTime->ulFrac / 10000);
            break;
        } // end switch on format

    return szTime;
    }
```

```
/* ---------------------------------------------------------------------- */

// This function fills in the SuTimeRef structure with the "best" relative
// and/or absolute time stamp available from the packet header and intra-packet
// header (if available).

EnI106Status I106_CALL_DECL
    vFillInTimeStruct(SuI106Ch10Header * psuHeader,
                      SuIntraPacketTS  * psuIntraPacketTS,
                      SuTimeRef        * psuTimeRef)
    {
    int iSecHdrTimeFmt;

    // Get the secondary header time format
    iSecHdrTimeFmt = psuHeader->ubyPacketFlags & I106CH10_PFLAGS_TIMEFMT_MASK;
    psuTimeRef->bRelTimeValid = bFALSE;
    psuTimeRef->bAbsTimeValid = bFALSE;

    // Set the relative time from the packet header
    vTimeArray2LLInt(psuHeader->aubyRefTime, &(psuTimeRef->uRelTime));
    psuTimeRef->bRelTimeValid = bTRUE;

    // If secondary header is available, use that time for absolute
    if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_SEC_HEADER) != 0)
        {
        switch(iSecHdrTimeFmt)
            {
            case I106CH10_PFLAGS_TIMEFMT_IRIG106:
                enI106_Ch4Binary2IrigTime((SuI106Ch4_Binary_Time *)psuHeader->aulTime,
&(psuTimeRef->suIrigTime));
                psuTimeRef->bAbsTimeValid = bTRUE;
                break;
            case I106CH10_PFLAGS_TIMEFMT_IEEE1588:
                enI106_IEEE15882IrigTime((SuIEEE1588_Time *)psuHeader->aulTime,
&(psuTimeRef->suIrigTime));
                psuTimeRef->bAbsTimeValid = bTRUE;
                break;
            default:
                //Currently reserved, should we have a default way to decode?
                break;
            } // end switch on secondary header time format
        } // end if

    // Now process values from the intra-packet headers if available
    if (psuIntraPacketTS != NULL)
        {
        // If relative time
        if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_IPTIMESRC) == 0)
            {
            vTimeArray2LLInt(psuIntraPacketTS->aubyIntPktTime, &(psuTimeRef-
>uRelTime));
            psuTimeRef->bRelTimeValid = bTRUE;
            }

        // else is absolute time
        else
            {
            switch(iSecHdrTimeFmt)
                {
                case I106CH10_PFLAGS_TIMEFMT_IRIG106:
                    enI106_Ch4Binary2IrigTime((SuI106Ch4_Binary_Time
```

```
*)psuIntraPacketTS, &(psuTimeRef->suIrigTime));
                    psuTimeRef->bAbsTimeValid = bTRUE;
                    break;
                case I106CH10_PFLAGS_TIMEFMT_IEEE1588:
                    enI106_IEEE15882IrigTime((SuIEEE1588_Time *)psuIntraPacketTS,
&(psuTimeRef->suIrigTime));
                    psuTimeRef->bAbsTimeValid = bTRUE;
                    break;
                default:
                    //Current reserved, should we have a default way to decode
                    break;
                } // end switch on intra-packet time format
            } // end else absolute time
        } // end if intra-packet time stamp exists

    return I106_OK;
    }



/* ------------------------------------------------------------------------ */

/* Return the equivalent in seconds past 12:00:00 a.m. Jan 1, 1970 GMT
   of the Greenwich Mean time and date in the exploded time structure `tm'.

   The standard mktime() has the annoying "feature" of assuming that the
   time in the tm structure is local time, and that it has to be corrected
   for local time zone. In this library time is assumed to be UTC and UTC
   only. To make sure no timezone correction is applied this time conversion
   routine was lifted from the standard C run time library source. Interestingly
   enough, this routine was found in the source for mktime().

   This function does always put back normalized values into the `tm' struct,
   parameter, including the calculated numbers for `tm->tm_yday',
   `tm->tm_wday', and `tm->tm_isdst'.

   Returns -1 if the time in the `tm' parameter cannot be represented
   as valid `time_t' number.
 */

// Number of leap years from 1970 to `y' (not including `y' itself).
#define nleap(y) (((y) - 1969) / 4 - ((y) - 1901) / 100 + ((y) - 1601) / 400)

// Nonzero if `y' is a leap year, else zero.
#define leap(y) (((y) % 4 == 0 && (y) % 100 != 0) || (y) % 400 == 0)

// Additional leapday in February of leap years.
#define leapday(m, y) ((m) == 1 && leap (y))

#define ADJUST_TM(tm_member, tm_carry, modulus) \
  if ((tm_member) < 0) { \
    tm_carry -= (1 - ((tm_member)+1) / (modulus)); \
    tm_member = (modulus-1) + (((tm_member)+1) % (modulus)); \
  } else if ((tm_member) >= (modulus)) { \
    tm_carry += (tm_member) / (modulus); \
    tm_member = (tm_member) % (modulus); \
  }

// Length of month `m' (0 .. 11)
#define monthlen(m, y) (ydays[(m)+1] - ydays[m] + leapday (m, y))


uint32_t I106_CALL_DECL mkgmtime(struct tm * psuTmTime)
```

```
{

// Accumulated number of days from 01-Jan up to start of current month.
static short ydays[] =
{
  0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365
};

int years, months, days, hours, minutes, seconds;

years   = psuTmTime->tm_year + 1900;  // year - 1900 -> year
months  = psuTmTime->tm_mon;          // 0..11
days    = psuTmTime->tm_mday - 1;     // 1..31 -> 0..30
hours   = psuTmTime->tm_hour;         // 0..23
minutes = psuTmTime->tm_min;          // 0..59
seconds = psuTmTime->tm_sec;          // 0..61 in ANSI C.

ADJUST_TM(seconds, minutes, 60)
ADJUST_TM(minutes, hours,   60)
ADJUST_TM(hours,   days,    24)
ADJUST_TM(months,  years,   12)

if (days < 0)
    do
        {
        if (--months < 0)
            {
            --years;
            months = 11;
            }
        days += monthlen(months, years);
        } while (days < 0);

else
    while (days >= monthlen(months, years))
        {
        days -= monthlen(months, years);
        if (++months >= 12)
            {
            ++years;
            months = 0;
            }
        } // end while

// Restore adjusted values in tm structure
psuTmTime->tm_year = years - 1900;
psuTmTime->tm_mon  = months;
psuTmTime->tm_mday = days + 1;
psuTmTime->tm_hour = hours;
psuTmTime->tm_min  = minutes;
psuTmTime->tm_sec  = seconds;

// Set `days' to the number of days into the year.
days += ydays[months] + (months > 1 && leap (years));
psuTmTime->tm_yday = days;

// Now calculate `days' to the number of days since Jan 1, 1970.
days = (unsigned)days + 365 * (unsigned)(years - 1970) +
       (unsigned)(nleap (years));
psuTmTime->tm_wday = ((unsigned)days + 4) % 7; /* Jan 1, 1970 was Thursday. */
psuTmTime->tm_isdst = 0;

if (years < 1970)
```

```
        return (uint32_t)-1;

    return (uint32_t)(86400L * days  + 3600L * hours + 60L * minutes + seconds);
    }

#ifdef __cplusplus
} // end namespace i106
#endif
```

Appendix A-5. i106_decode_time.h

```
/****************************************************************************

 i106_decode_time.h -

 Copyright (c) 2005 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#ifndef _I106_DECODE_TIME_H
#define _I106_DECODE_TIME_H

//#include "irig106ch10.h"

#ifdef __cplusplus
namespace Irig106 {
extern "C" {
#endif

/*
 * Macros and definitions
 * ----------------------
 */

typedef enum
    {
    I106_TIMEFMT_IRIG_B      =  0x00,
    I106_TIMEFMT_IRIG_A      =  0x01,
    I106_TIMEFMT_IRIG_G      =  0x02,
    I106_TIMEFMT_INT_RTC     =  0x03,
    I106_TIMEFMT_GPS_UTC     =  0x04,
    I106_TIMEFMT_GPS_NATIVE  =  0x05,
    } EnI106TimeFmt;
```

```
typedef enum
    {
    I106_TIMESRC_INTERNAL      =  0x00,
    I106_TIMESRC_EXTERNAL      =  0x01,
    I106_TIMESRC_INTERNAL_RMM  =  0x02,
    I106_TIMESRC_NONE          =  0x0F
    } EnI106TimeSrc;


/*
 * Data structures
 * ---------------
 */

#if defined(_MSC_VER)
#pragma pack(push)
#pragma pack(1)
#endif

/* Time Format 1 */

/// Time Format 1 Channel Specific Data Word
typedef struct
    {
    uint32_t    uTimeSrc    :  4;      // Time source
    uint32_t    uTimeFmt    :  4;      // Time format
    uint32_t    bLeapYear   :  1;      // Leap year
    uint32_t    uDateFmt    :  1;      // Date format
    uint32_t    uReserved2  :  2;
    uint32_t    uReserved3  : 20;
#if !defined(__GNUC__)
    } SuTimeF1_ChanSpec;
#else
    } __attribute__ ((packed)) SuTimeF1_ChanSpec;
#endif

// Time message - Day format
typedef struct
    {
    uint16_t    uTmn        :  4;      // Tens of milliseconds
    uint16_t    uHmn        :  4;      // Hundreds of milliseconds
    uint16_t    uSn         :  4;      // Units of seconds
    uint16_t    uTSn        :  3;      // Tens of seconds
    uint16_t    Reserved1   :  1;      // 0

    uint16_t    uMn         :  4;      // Units of minutes
    uint16_t    uTMn        :  3;      // Tens of minutes
    uint16_t    Reserved2   :  1;      // 0
    uint16_t    uHn         :  4;      // Units of hours
    uint16_t    uTHn        :  2;      // Tens of Hours
    uint16_t    Reserved3   :  2;      // 0

    uint16_t    uDn         :  4;      // Units of day number
    uint16_t    uTDn        :  4;      // Tens of day number
    uint16_t    uHDn        :  2;      // Hundreds of day number
    uint16_t    Reserved4   :  6;      // 0
#if !defined(__GNUC__)
    } SuTime_MsgDayFmt;
#else
    } __attribute__ ((packed)) SuTime_MsgDayFmt;
#endif

// Time message - DMY format
```

```
typedef struct
    {
    uint16_t    uTmn        :  4;      // Tens of milliseconds
    uint16_t    uHmn        :  4;      // Hundreds of milliseconds
    uint16_t    uSn         :  4;      // Units of seconds
    uint16_t    uTSn        :  3;      // Tens of seconds
    uint16_t    Reserved1   :  1;      // 0

    uint16_t    uMn         :  4;      // Units of minutes
    uint16_t    uTMn        :  3;      // Tens of minutes
    uint16_t    Reserved2   :  1;      // 0
    uint16_t    uHn         :  4;      // Units of hours
    uint16_t    uTHn        :  2;      // Tens of Hours
    uint16_t    Reserved3   :  2;      // 0

    uint16_t    uDn         :  4;      // Units of day number
    uint16_t    uTDn        :  4;      // Tens of day number
    uint16_t    uOn         :  4;      // Units of month number
    uint16_t    uTOn        :  1;      // Tens of month number
    uint16_t    Reserved4   :  3;      // 0

    uint16_t    uYn         :  4;      // Units of year number
    uint16_t    uTYn        :  4;      // Tens of year number
    uint16_t    uHYn        :  4;      // Hundreds of year number
    uint16_t    uOYn        :  2;      // Thousands of year number
    uint16_t    Reserved5   :  2;      // 0
#if !defined(__GNUC__)
    } SuTime_MsgDmyFmt;
#else
    } __attribute__ ((packed)) SuTime_MsgDmyFmt;
#endif

// Time message Format 1 structure
typedef struct
    {
    SuTimeF1_ChanSpec   suChanSpec;
    union
        {
        SuTime_MsgDayFmt    suDayFmt;
        SuTime_MsgDmyFmt    suDmyFmt;
        } suMsg;
    } SuMsgTimeF1;


/* Time Format 2 */

/// Time Format 2 Channel Specific Data Word
typedef struct
    {
    uint32_t    uTimeStatus :  4;      // Time status
    uint32_t    uTimeFmt    :  4;      // Network time format
    uint32_t    uReserved   : 24;
#if !defined(__GNUC__)
    } SuTimeF2_ChanSpec;
#else
    } __attribute__ ((packed)) SuTimeF2_ChanSpec;
#endif

// Time message Format 2 structure
typedef struct
    {
    uint32_t    uSeconds;               // Integer time value
    utin32_t    uSubseconds;            // Fractional part of time value
```

```
#if !defined(__GNUC__)
    } SuTimeF2_Data;
#else
    } __attribute__ ((packed)) SuTimeF2_Data;
#endif



#if defined(_MSC_VER)
#pragma pack(pop)
#endif

/*
 * Function Declaration
 * --------------------
 */

EnI106Status I106_CALL_DECL
    enI106_Decode_TimeF1(SuI106Ch10Header  * psuHeader,
                         void              * pvBuff,
                         SuIrig106Time     * psuTime);

void I106_CALL_DECL
    enI106_Decode_TimeF1_Buff(int                iDateFmt,
                              int                bLeapYear,
                              void             * pvTimeBuff,
                              SuIrig106Time    * psuTime);

EnI106Status I106_CALL_DECL
    enI106_Encode_TimeF1(SuI106Ch10Header  * psuHeader,
                         unsigned int        uTimeSrc,
                         unsigned int        uFmtTime,
                         unsigned int        uFmtDate,
                         SuIrig106Time     * psuTime,
                         void              * pvBuffTimeF1);

#ifdef __cplusplus
}
}
#endif

#endif
```

Appendix A-6. i106_decode_time.c

```
/****************************************************************************

 i106_time.c -

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include <time.h>

#include "stdint.h"
#include "irig106ch10.h"
#include "i106_time.h"
#include "i106_decode_time.h"

#ifdef __cplusplus
namespace Irig106 {
#endif

/*
 * Macros and definitions
 * ----------------------
 */


/*
 * Data structures
 * ---------------
 */
```

```
/*
 * Module data
 * -----------
 */

static SuTimeRef    m_asuTimeRef[MAX_HANDLES]; // Relative / absolute time reference

/*
 * Function Declaration
 * --------------------
 */


/* ------------------------------------------------------------------------ */

// Update the current reference time value
EnI106Status I106_CALL_DECL
    enI106_SetRelTime(int              iI106Ch10Handle,
                      SuIrig106Time  * psuTime,
                      uint8_t          abyRelTime[])
    {

    // Save the absolute time value
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs = psuTime->ulSecs;
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac = psuTime->ulFrac;
    m_asuTimeRef[iI106Ch10Handle].suIrigTime.enFmt  = psuTime->enFmt;

    // Save the relative (i.e. the 10MHz counter) value
    m_asuTimeRef[iI106Ch10Handle].uRelTime          = 0;
    memcpy((char *)&(m_asuTimeRef[iI106Ch10Handle].uRelTime),
           (char *)&abyRelTime[0], 6);

    return I106_OK;
    }



/* ------------------------------------------------------------------------ */

// Take a 6 byte relative time value (like the one in the IRIG header) and
// turn it into a real time based on the current reference IRIG time.

EnI106Status I106_CALL_DECL
    enI106_Rel2IrigTime(int              iI106Ch10Handle,
                        uint8_t          abyRelTime[],
                        SuIrig106Time  * psuTime)
    {
    int64_t         llRelTime;
    EnI106Status    enStatus;

    // Convert 6 byte time array to 16 bit int. This only works for
    // positive time, but that shouldn't be a problem
    llRelTime = 0L;
    memcpy(&llRelTime, &abyRelTime[0], 6);

    enStatus = enI106_RelInt2IrigTime(iI106Ch10Handle, llRelTime, psuTime);

    return enStatus;
    }
```

```
/* ---------------------------------------------------------------------- */

// Take a 64 bit relative time value and turn it into a real time based on
// the current reference IRIG time.

EnI106Status I106_CALL_DECL
    enI106_RelInt2IrigTime(int               iI106Ch10Handle,
                           int64_t           llRelTime,
                           SuIrig106Time   * psuTime)
    {
    int64_t         uTimeDiff;
    int64_t         lFracDiff;
    int64_t         lSecDiff;

    int64_t         lSec;
    int64_t         lFrac;


    // Figure out the relative time difference
    uTimeDiff = llRelTime - m_asuTimeRef[iI106Ch10Handle].uRelTime;
    lSecDiff  = uTimeDiff / 10000000;
    lFracDiff = uTimeDiff % 10000000;

    lSec      = m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs + lSecDiff;
    lFrac     = m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac + lFracDiff;

    // This seems a bit extreme but it's defensive programming
    while (lFrac < 0)
        {
        lFrac += 10000000;
        lSec  -= 1;
        }

    while (lFrac >= 10000000)
        {
        lFrac -= 10000000;
        lSec  += 1;
        }

    // Now add the time difference to the last IRIG time reference
    psuTime->ulFrac = (unsigned long)lFrac;
    psuTime->ulSecs = (unsigned long)lSec;
    psuTime->enFmt  = m_asuTimeRef[iI106Ch10Handle].suIrigTime.enFmt;

    return I106_OK;
    }



/* ---------------------------------------------------------------------- */

// Take a real clock time and turn it into a 6 byte relative time.

EnI106Status I106_CALL_DECL
    enI106_Irig2RelTime(int              iI106Ch10Handle,
                        SuIrig106Time  * psuTime,
                        uint8_t          abyRelTime[])
    {
    int64_t         llDiff;
    int64_t         llNewRel;

    // Calculate time difference (LSB = 100 nSec) between the passed time
```

```
    // and the time reference
    llDiff =
         (int64_t)(+ psuTime->ulSecs -
m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulSecs) * 10000000 +
         (int64_t)(+ psuTime->ulFrac -
m_asuTimeRef[iI106Ch10Handle].suIrigTime.ulFrac);

    // Add this amount to the reference
    llNewRel = m_asuTimeRef[iI106Ch10Handle].uRelTime + llDiff;

    // Now convert this to a 6 byte relative time
    memcpy((char *)&abyRelTime[0],
           (char *)&(llNewRel), 6);

    return I106_OK;
    }

/* ---------------------------------------------------------------------- */

// Take a Irig Ch4 time value (like the one in a secondary IRIG header) and
// turn it into an Irig106 time

EnI106Status I106_CALL_DECL
    enI106_Ch4Binary2IrigTime(SuI106Ch4_Binary_Time * psuCh4BinaryTime,
                              SuIrig106Time         * psuIrig106Time)
{
    psuIrig106Time->ulSecs = (unsigned long)
        ( (double)psuCh4BinaryTime->uHighBinTime * CH4BINARYTIME_HIGH_LSB_SEC
        + (unsigned long)psuCh4BinaryTime->uLowBinTime * CH4BINARYTIME_LOW_LSB_SEC );
    psuIrig106Time->ulFrac = (unsigned long)psuCh4BinaryTime->uUSecs *
_100_NANO_SEC_IN_MICRO_SEC;

    return I106_OK;
}

/* ---------------------------------------------------------------------- */

// Take a IEEE-1588 time value (like the one in a secondary IRIG header) and
// turn it into an Irig106 time
EnI106Status I106_CALL_DECL
    enI106_IEEE15882IrigTime(SuIEEE1588_Time * psuIEEE1588Time,
                              SuIrig106Time  * psuIrig106Time)
{
    psuIrig106Time->ulSecs = (unsigned long)psuIEEE1588Time->uSeconds;
    //Convert 'nanoseconds' to '100 nanoseconds'
    psuIrig106Time->ulFrac = (unsigned long)psuIEEE1588Time->uNanoSeconds/100;

    return I106_OK;
}


/* ---------------------------------------------------------------------- */

// Warning - array to int / int to array functions are little endian only!

// Create a 6 byte array value from a 64 bit int relative time

void I106_CALL_DECL
    vLLInt2TimeArray(int64_t * pllRelTime,
                     uint8_t   abyRelTime[])
    {
    memcpy((char *)abyRelTime, (char *)pllRelTime, 6);
    return;
```

```
    }


/* ---------------------------------------------------------------------- */

// Create a 64 bit int relative time from 6 byte array value

void I106_CALL_DECL
    vTimeArray2LLInt(uint8_t   abyRelTime[],
                     int64_t * pllRelTime)
    {
    *pllRelTime = 0L;
    memcpy((char *)pllRelTime, (char *)abyRelTime, 6);
    return;
    }



/* ---------------------------------------------------------------------- */

// Read the data file from the current position to try to determine a valid
// relative time to clock time from a time packet.

EnI106Status I106_CALL_DECL
    enI106_SyncTime(int     iI106Ch10Handle,
                    int     bRequireSync,   // Require external time source sync
                    int     iTimeLimit)     // Max time to look in seconds
    {
    int64_t             llCurrOffset;
    int64_t             llTimeLimit;
    int64_t             llCurrTime;
    EnI106Status        enStatus;
    EnI106Status        enRetStatus;
    SuI106Ch10Header    suI106Hdr;
    SuIrig106Time       suTime;
    unsigned long       ulBuffSize = 0;
    void              * pvBuff = NULL;
    SuTimeF1_ChanSpec * psuChanSpecTime = NULL;

    // Get and save the current file position
    enStatus = enI106Ch10GetPos(iI106Ch10Handle, &llCurrOffset);
    if (enStatus != I106_OK)
        return enStatus;

    // Set the file to the start
    //enStatus = enI106Ch10SetPos(iI106Ch10Handle, 0);
    //if (enStatus != I106_OK)
    //    return enStatus;

    // Read the next header
    enStatus = enI106Ch10ReadNextHeaderFile(iI106Ch10Handle, &suI106Hdr);
    if (enStatus == I106_EOF)
        return I106_TIME_NOT_FOUND;

    if (enStatus != I106_OK)
        return enStatus;

    // Calculate the time limit if there is one
    if (iTimeLimit > 0)
        {
        vTimeArray2LLInt(suI106Hdr.aubyRefTime, &llTimeLimit);
        llTimeLimit = llTimeLimit + (int64_t)iTimeLimit * (int64_t)10000000;
        }
```

```
else
    llTimeLimit = 0;

// Loop, looking for appropriate time message
while (bTRUE)
    {

    // See if we've passed our time limit
    if (llTimeLimit > 0)
        {
        vTimeArray2LLInt(suI106Hdr.aubyRefTime, &llCurrTime);
        if (llTimeLimit < llCurrTime)
            {
            enRetStatus = I106_TIME_NOT_FOUND;
            break;
            }
        } // end if there is a time limit

    // If IRIG time type then process it
    if (suI106Hdr.ubyDataType == I106CH10_DTYPE_IRIG_TIME)
        {

        // Read header OK, make buffer for time message
        if (ulBuffSize < suI106Hdr.ulPacketLen)
            {
            pvBuff          = realloc(pvBuff, suI106Hdr.ulPacketLen);
            psuChanSpecTime = (SuTimeF1_ChanSpec *)pvBuff;
            ulBuffSize      = suI106Hdr.ulPacketLen;
            }

        // Read the data buffer
        enStatus = enI106Ch10ReadData(iI106Ch10Handle, ulBuffSize, pvBuff);
        if (enStatus != I106_OK)
            {
            enRetStatus = I106_TIME_NOT_FOUND;
            break;
            }

        // If external sync OK then decode it and set relative time
        if ((bRequireSync == bFALSE) || (psuChanSpecTime->uTimeSrc == 1))
            {
            enI106_Decode_TimeF1(&suI106Hdr, pvBuff, &suTime);
            enI106_SetRelTime(iI106Ch10Handle, &suTime, suI106Hdr.aubyRefTime);
            enRetStatus = I106_OK;
            break;
            }
        } // end if IRIG time message

    // Read the next header and try again
    enStatus = enI106Ch10ReadNextHeaderFile(iI106Ch10Handle, &suI106Hdr);
    if (enStatus == I106_EOF)
        {
        enRetStatus = I106_TIME_NOT_FOUND;
        break;
        }

    if (enStatus != I106_OK)
        {
        enRetStatus = enStatus;
        break;
        }

    } // end while looping looking for time message
```

```
    // Restore file position
    enStatus = enI106Ch10SetPos(iI106Ch10Handle, llCurrOffset);
    if (enStatus != I106_OK)
        {
        enRetStatus = enStatus;
        }

    // Return the malloc'ed memory
    free(pvBuff);

    return enRetStatus;
    }


/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106Ch10SetPosToIrigTime(int iHandle, SuIrig106Time * psuSeekTime)
    {
    uint8_t             abySeekTime[6];
    int64_t             llSeekTime;
    SuInOrderIndex    * psuIndex = &g_suI106Handle[iHandle].suInOrderIndex;
    int                 iUpperLimit;
    int                 iLowerLimit;
    int                 iSearchLoopIdx;

    // If there is no index in memory then barf
    if (psuIndex->enSortStatus != enSorted)
        return I106_NO_INDEX;

    // We have an index so do a binary search for time

    // Convert clock time to 10 MHz count
    enI106_Irig2RelTime(iHandle, psuSeekTime, abySeekTime);
    vTimeArray2LLInt(abySeekTime, &llSeekTime);

    // Check time bounds
    if (llSeekTime < psuIndex->asuIndex[0].llTime)
        {
        enI106Ch10FirstMsg(iHandle);
        return I106_TIME_NOT_FOUND;
        };

    if (llSeekTime > psuIndex->asuIndex[psuIndex->iArrayUsed].llTime)
        {
        enI106Ch10LastMsg(iHandle);
        return I106_TIME_NOT_FOUND;
        };

    // If we don't already have it, figure out how many search steps
    if (psuIndex->iNumSearchSteps == 0)
        {
        iUpperLimit = 1;
        while (iUpperLimit < psuIndex->iArrayUsed)
            {
            iUpperLimit *= 2;
            psuIndex->iNumSearchSteps++;
            }
        } // end if no search steps

    // Loop prescribed number of times
    iLowerLimit = 0;
```

```
    iUpperLimit = psuIndex->iArrayUsed-1;
    psuIndex->iArrayCurr = (iUpperLimit - iLowerLimit) / 2;
    for (iSearchLoopIdx = 0;
         iSearchLoopIdx < psuIndex->iNumSearchSteps;
         iSearchLoopIdx++)
        {
        if      (psuIndex->asuIndex[psuIndex->iArrayCurr].llTime > llSeekTime)
            iUpperLimit = (iUpperLimit - iLowerLimit) / 2;
        else if (psuIndex->asuIndex[psuIndex->iArrayCurr].llTime < llSeekTime)
            iLowerLimit = (iUpperLimit - iLowerLimit) / 2;
        else
            break;
        }

    return I106_OK;
    }


/* ---------------------------------------------------------------------- */

// General purpose time utilities
// ------------------------------

// Convert IRIG time into an appropriate string

char * IrigTime2String(SuIrig106Time * psuTime)
    {
    static char     szTime[30];
    struct tm     * psuTmTime;

    // Convert IRIG time into it's components
    psuTmTime = gmtime((time_t *)&(psuTime->ulSecs));

    // Make the appropriate string
    switch (psuTime->enFmt)
        {
        // Year / Month / Day format ("2008/02/29 12:34:56.789")
        case I106_DATEFMT_DMY :
            sprintf(szTime, "%4.4i/%2.2i/%2.2i %2.2i:%2.2i:%2.2i.%3.3i",
                psuTmTime->tm_year + 1900,
                psuTmTime->tm_mon + 1,
                psuTmTime->tm_mday,
                psuTmTime->tm_hour,
                psuTmTime->tm_min,
                psuTmTime->tm_sec,
                psuTime->ulFrac / 10000);
            break;

        // Day of the Year format ("001:12:34:56.789")
        case I106_DATEFMT_DAY :
        default :
            sprintf(szTime, "%3.3i:%2.2i:%2.2i:%2.2i.%3.3i",
                psuTmTime->tm_yday+1,
                psuTmTime->tm_hour,
                psuTmTime->tm_min,
                psuTmTime->tm_sec,
                psuTime->ulFrac / 10000);
            break;
        } // end switch on format

    return szTime;
    }
```

```
/* ---------------------------------------------------------------------- */

// This function fills in the SuTimeRef structure with the "best" relative
// and/or absolute time stamp available from the packet header and intra-packet
// header (if available).

EnI106Status I106_CALL_DECL
    vFillInTimeStruct(SuI106Ch10Header * psuHeader,
                      SuIntraPacketTS  * psuIntraPacketTS,
                      SuTimeRef        * psuTimeRef)
    {
    int iSecHdrTimeFmt;

    // Get the secondary header time format
    iSecHdrTimeFmt = psuHeader->ubyPacketFlags & I106CH10_PFLAGS_TIMEFMT_MASK;
    psuTimeRef->bRelTimeValid = bFALSE;
    psuTimeRef->bAbsTimeValid = bFALSE;

    // Set the relative time from the packet header
    vTimeArray2LLInt(psuHeader->aubyRefTime, &(psuTimeRef->uRelTime));
    psuTimeRef->bRelTimeValid = bTRUE;

    // If secondary header is available, use that time for absolute
    if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_SEC_HEADER) != 0)
        {
        switch(iSecHdrTimeFmt)
            {
            case I106CH10_PFLAGS_TIMEFMT_IRIG106:
                enI106_Ch4Binary2IrigTime((SuI106Ch4_Binary_Time *)psuHeader->aulTime,
&(psuTimeRef->suIrigTime));
                psuTimeRef->bAbsTimeValid = bTRUE;
                break;
            case I106CH10_PFLAGS_TIMEFMT_IEEE1588:
                enI106_IEEE15882IrigTime((SuIEEE1588_Time *)psuHeader->aulTime,
&(psuTimeRef->suIrigTime));
                psuTimeRef->bAbsTimeValid = bTRUE;
                break;
            default:
                //Currently reserved, should we have a default way to decode?
                break;
            } // end switch on secondary header time format
        } // end if

    // Now process values from the intra-packet headers if available
    if (psuIntraPacketTS != NULL)
        {
        // If relative time
        if ((psuHeader->ubyPacketFlags & I106CH10_PFLAGS_IPTIMESRC) == 0)
            {
            vTimeArray2LLInt(psuIntraPacketTS->aubyIntPktTime, &(psuTimeRef-
>uRelTime));
            psuTimeRef->bRelTimeValid = bTRUE;
            }

        // else is absolute time
        else
            {
            switch(iSecHdrTimeFmt)
                {
                case I106CH10_PFLAGS_TIMEFMT_IRIG106:
                    enI106_Ch4Binary2IrigTime((SuI106Ch4_Binary_Time
```

```
*)psuIntraPacketTS, &(psuTimeRef->suIrigTime));
                    psuTimeRef->bAbsTimeValid = bTRUE;
                    break;
                case I106CH10_PFLAGS_TIMEFMT_IEEE1588:
                    enI106_IEEE15882IrigTime((SuIEEE1588_Time *)psuIntraPacketTS,
&(psuTimeRef->suIrigTime));
                    psuTimeRef->bAbsTimeValid = bTRUE;
                    break;
                default:
                    //Current reserved, should we have a default way to decode
                    break;
                } // end switch on intra-packet time format
            } // end else absolute time
        } // end if intra-packet time stamp exists

    return I106_OK;
    }



/* ------------------------------------------------------------------------ */

/* Return the equivalent in seconds past 12:00:00 a.m. Jan 1, 1970 GMT
   of the Greenwich Mean time and date in the exploded time structure `tm'.

   The standard mktime() has the annoying "feature" of assuming that the
   time in the tm structure is local time, and that it has to be corrected
   for local time zone. In this library time is assumed to be UTC and UTC
   only. To make sure no timezone correction is applied this time conversion
   routine was lifted from the standard C run time library source. Interestingly
   enough, this routine was found in the source for mktime().

   This function does always put back normalized values into the `tm' struct,
   parameter, including the calculated numbers for `tm->tm_yday',
   `tm->tm_wday', and `tm->tm_isdst'.

   Returns -1 if the time in the `tm' parameter cannot be represented
   as valid `time_t' number.
 */

// Number of leap years from 1970 to `y' (not including `y' itself).
#define nleap(y) (((y) - 1969) / 4 - ((y) - 1901) / 100 + ((y) - 1601) / 400)

// Nonzero if `y' is a leap year, else zero.
#define leap(y) (((y) % 4 == 0 && (y) % 100 != 0) || (y) % 400 == 0)

// Additional leapday in February of leap years.
#define leapday(m, y) ((m) == 1 && leap (y))

#define ADJUST_TM(tm_member, tm_carry, modulus) \
  if ((tm_member) < 0) { \
    tm_carry -= (1 - ((tm_member)+1) / (modulus)); \
    tm_member = (modulus-1) + (((tm_member)+1) % (modulus)); \
  } else if ((tm_member) >= (modulus)) { \
    tm_carry += (tm_member) / (modulus); \
    tm_member = (tm_member) % (modulus); \
  }

// Length of month `m' (0 .. 11)
#define monthlen(m, y) (ydays[(m)+1] - ydays[m] + leapday (m, y))


uint32_t I106_CALL_DECL mkgmtime(struct tm * psuTmTime)
```

```
{

// Accumulated number of days from 01-Jan up to start of current month.
static short ydays[] =
{
  0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365
};

int years, months, days, hours, minutes, seconds;

years   = psuTmTime->tm_year + 1900;  // year - 1900 -> year
months  = psuTmTime->tm_mon;          // 0..11
days    = psuTmTime->tm_mday - 1;     // 1..31 -> 0..30
hours   = psuTmTime->tm_hour;         // 0..23
minutes = psuTmTime->tm_min;          // 0..59
seconds = psuTmTime->tm_sec;          // 0..61 in ANSI C.

ADJUST_TM(seconds, minutes, 60)
ADJUST_TM(minutes, hours,   60)
ADJUST_TM(hours,   days,    24)
ADJUST_TM(months,  years,   12)

if (days < 0)
    do
        {
        if (--months < 0)
            {
            --years;
            months = 11;
            }
        days += monthlen(months, years);
        } while (days < 0);

else
    while (days >= monthlen(months, years))
        {
        days -= monthlen(months, years);
        if (++months >= 12)
            {
            ++years;
            months = 0;
            }
        } // end while

// Restore adjusted values in tm structure
psuTmTime->tm_year = years - 1900;
psuTmTime->tm_mon  = months;
psuTmTime->tm_mday = days + 1;
psuTmTime->tm_hour = hours;
psuTmTime->tm_min  = minutes;
psuTmTime->tm_sec  = seconds;

// Set `days' to the number of days into the year.
days += ydays[months] + (months > 1 && leap (years));
psuTmTime->tm_yday = days;

// Now calculate `days' to the number of days since Jan 1, 1970.
days = (unsigned)days + 365 * (unsigned)(years - 1970) +
       (unsigned)(nleap (years));
psuTmTime->tm_wday = ((unsigned)days + 4) % 7; /* Jan 1, 1970 was Thursday. */
psuTmTime->tm_isdst = 0;

if (years < 1970)
```

```
        return (uint32_t)-1;

    return (uint32_t)(86400L * days  + 3600L * hours + 60L * minutes + seconds);
    }

#ifdef __cplusplus
} // end namespace i106
#endif
```

Appendix A-7. i106_data_stream.h

```
/****************************************************************************

 i106_data_stream.h - A module to decode Chapter 10 UDP data streaming

 Copyright (c) 2011 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#ifndef _I106_DATA_STREAMING_H
#define _I106_DATA_STREAMING_H

#ifdef __cplusplus
namespace Irig106 {
extern "C" {
#endif

/*
 * Macros and definitions
 * ----------------------
 */



/*
 * Data structures
 * ---------------
 */

#if defined(_MSC_VER)
#pragma pack(push)
#pragma pack(1)
#endif
```

```
/* UDP Streaming Format 1 */

/// UDP Transfer Header Format 1 - Non-segmented
typedef struct
    {
    uint32_t    uFormat         :  4;   ///< Streaming version
    uint32_t    uMsgType        :  4;   ///< Type of message
    uint32_t    uUdpSeqNum      : 24;   ///< UDP sequence number
    uint8_t     achData[1];             ///< Start of Ch 10 data packet
#if !defined(__GNUC__)
    } SuUDP_Transfer_Header_F1_NonSeg;
#else
    } __attribute__ ((packed)) SuUDP_Transfer_Header_F1_NonSeg;
#endif

enum { UDP_Transfer_Header_NonSeg_Len = sizeof(SuUDP_Transfer_Header_NonSeg) - 1 };

// UDP Transfer Header Format 1 - Segmented
typedef struct
    {
    uint32_t    uFormat         :  4;   ///< Streaming version
    uint32_t    uMsgType        :  4;   ///< Type of message
    uint32_t    uUdpSeqNum      : 24;   ///< UDP sequence number
    uint32_t    uChID           : 16;   ///< Channel ID
    uint32_t    uChanSeqNum     :  8;   ///< Channel sequence number
    uint32_t    uReserved       :  8;
    uint32_t    uSegmentOffset;         ///< Segment offset
    uint8_t     achData[1];             ///< Start of Ch 10 data packet
#if !defined(__GNUC__)
    } SuUDP_Transfer_Header_F1_Seg;
#else
    } __attribute__ ((packed)) SuUDP_Transfer_Header_F1_Seg;
#endif

enum { UDP_Transfer_Header_Seg_Len = sizeof(SuUDP_Transfer_Header_F1_Seg) - 1 };

/* UDP Streaming Format 2 */

/// UDP Transfer Header Format 2
typedef struct
    {
    uint32_t    uFormat         :  4;   ///< Streaming version
    uint32_t    uMsgType        :  4;   ///< Type of message
    uint32_t    uUdpSeqNum      : 24;   ///< UDP sequence number
    uint32_t    uPacketSize     : 24;   ///< Packet size in bytes
    uint32_t    uSegmentOffset1 :  8;
    uint32_t    uChanID;        : 16;
    uint32_t    uSegmentOffset2 : 16;
    uint8_t     achData[1];             // Start of Ch 10 data packet
#if !defined(__GNUC__)
    } SuUDP_Transfer_Header_F2;
#else
    } __attribute__ ((packed)) SuUDP_Transfer_Header_F2;
#endif


/* UDP Streaming Format 3 */

/// UDP Transfer Header Format 3
typedef struct
    {
    uint32_t    uFormat         :  4;   ///< Streaming version
    uint32_t    uSrcIdLen       :  4;   ///< Type of message
```

```
    uint32_t    uReserved       :  8;
    uint32_t    uPktStartOffset : 16;   ///< Start of Ch 11 packet
    uint32_t    uUdpSeqNum      : 24;   ///< UDP sequence number
    uint32_t    uSrcId          :  8;   ///< Source identifier
    uint8_t     achData[1];             ///< Start of Ch 10 data packet
#if !defined(__GNUC__)
    } SuUDP_Transfer_Header_F3;
#else
    } __attribute__ ((packed)) SuUDP_Transfer_Header_F3;
#endif


#if defined(_MSC_VER)
#pragma pack(pop)
#endif

/*
 * Function Declaration
 * --------------------
 */


// Open / Close

EnI106Status I106_CALL_DECL
    enI106_OpenNetStreamRead(int                iHandle,
                             uint16_t           uPort);

EnI106Status I106_CALL_DECL
    enI106_OpenNetStreamWrite(int iHandle,
                              uint32_t          uIpAddress,
                              uint16_t          uUdpPort);

EnI106Status I106_CALL_DECL
    enI106_CloseNetStream(int                iHandle);


// Read
// ----

int I106_CALL_DECL
    enI106_ReadNetStream(int            iHandle,
                         void         * pvBuffer,
                         uint32_t       uBuffSize);

// Manipulate receive buffer
// -------------------------

EnI106Status I106_CALL_DECL
    enI106_DumpNetStream(int iHandle);

EnI106Status I106_CALL_DECL
    enI106_MoveReadPointer(int iHandle, long iRelOffset);

// Write
// -----

EnI106Status I106_CALL_DECL
    enI106_WriteNetStream(
            int           iHandle,
            void        * pvBuffer,
            uint32_t      uBuffSize);
```

```
EnI106Status I106_CALL_DECL
    enI106_WriteNetNonSegmented(
            int           iHandle,
            void        * pvBuffer,
            uint32_t      uBuffSize);

EnI106Status I106_CALL_DECL
    enI106_WriteNetSegmented(
            int           iHandle,
            void        * pvBuffer,
            uint32_t      uBuffSize);

#ifdef __cplusplus
}
}
#endif

#endif
```

Appendix A-8. i106_data_stream.c

```
/****************************************************************************

 i106_data_stream.c -

 Copyright (c) 2011 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#if defined(__GNUC__)
#define SOCKET            int
#define INVALID_SOCKET    -1
#define SOCKET_ERROR      -1
#define SOCKADDR          struct sockaddr
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#endif

#include <assert.h>

#if defined(_WIN32)
#define WIN32_LEAN_AND_MEAN             // Exclude rarely-used stuff from Windows
headers
#include <windows.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <mswsock.h>
#endif
```

```
#include "config.h"
#include "stdint.h"

#include "irig106ch10.h"
#include "i106_time.h"
#include "i106_data_stream.h"

#ifdef __cplusplus
namespace Irig106 {
#endif

/*
 * Macros and definitions
 * ----------------------
 */

// Make a min() function
#if defined(_WIN32)
#define MIN(X, Y)   min(X, Y)
#else
#define MIN(X, Y) (((X) < (Y)) ? (X) : (Y))
#endif

#define RCV_BUFFER_START_SIZE   32768
#define MAX_UDP_WRITE_SIZE      32726   // From Chapter 10.3.9.1.3


/*
 * Data structures
 * ---------------
 */

/// Data structure for IRIG 106 network handle
typedef struct
    {
    EnI106Ch10Mode      enNetMode;
    SOCKET              suIrigSocket;
    unsigned int        uUdpSeqNum;
    // Receive buffer stuff
    char              * pchRcvBuffer;
    unsigned long       ulRcvBufferLen;
    unsigned long       ulRcvBufferDataLen;
    int                 bBufferReady;
    unsigned long       ulBufferPosIdx;
    int                 bGotFirstSegment;
    // Transmit buffer stuff
    struct sockaddr_in  suSendIpAddress;
    uint16_t            uSendPort;
    unsigned int        uMaxUdpSize;    // Max size of Ch 10 message(s)
    } SuI106Ch10NetHandle;

/*
 * Module data
 * -----------
 */

static int                  m_bHandlesInited = bFALSE;
static SuI106Ch10NetHandle  m_suNetHandle[MAX_HANDLES];


/*
 * Function Declaration
 * --------------------
```

```
 */


/* ---------------------------------------------------------------------- */


// Open / Close

/// Open an IRIG 106 Live Data Streaming receive socket
EnI106Status I106_CALL_DECL
    enI106_OpenNetStreamRead(int iHandle, uint16_t uPort)
    {
    int                     iIdx;
    int                     iResult;
    struct sockaddr_in      ServerAddr;
#if defined(_MSC_VER)
    WORD                    wVersionRequested;
    WSADATA                 wsaData;
#endif

    // Initialize handle data if necessary
    if (m_bHandlesInited == bFALSE)
        {
        for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
            {
            m_suNetHandle[iIdx].enNetMode  = I106_CLOSED;
            }
        m_bHandlesInited = bTRUE;
        } // end if file handles not inited yet


#ifdef MULTICAST
    int                     iInterfaceIdx;
    int                     iNumInterfaces;

    struct in_addr          NetInterfaces[10];
    struct in_addr          LocalInterfaceAddr;
    struct in_addr          LocalInterfaceMask;
    struct in_addr          IrigMulticastGroup;
#endif

#if defined(_MSC_VER)
    // Initialize WinSock, request version 2.2
    wVersionRequested = MAKEWORD(2, 2);
    iResult = WSAStartup(wVersionRequested, &wsaData);

    if (iResult != 0)
        {
//      printf("Unable to initialize Winsock 2.2\n");
        return I106_OPEN_ERROR;
        }
#endif

    // Create a socket for listening to UDP
    m_suNetHandle[iHandle].suIrigSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (m_suNetHandle[iHandle].suIrigSocket == INVALID_SOCKET)
        {
//        printf("socket() failed with error: %ld\n", WSAGetLastError());
#if defined(_MSC_VER)
        WSACleanup();
#endif
        return I106_OPEN_ERROR;
        }
```

```
    // Bind to any local address
    ServerAddr.sin_family      = AF_INET;
    ServerAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    ServerAddr.sin_port        = htons(uPort);

    iResult = bind(m_suNetHandle[iHandle].suIrigSocket, (SOCKADDR*) &ServerAddr,
sizeof(ServerAddr));
    if (iResult == SOCKET_ERROR)
        {
//        printf("bind() failed with error: %ld\n", WSAGetLastError());
#if defined(_MSC_VER)
        closesocket(m_suNetHandle[iHandle].suIrigSocket);
        WSACleanup();
#else
        close(m_suNetHandle[iHandle].suIrigSocket);
#endif
        return I106_OPEN_ERROR;
        }

#ifdef MULTICAST
    // Put the appropriate interface into multicast receive mode
    iNumInterfaces = GetInterfaces(NetInterfaces, 10);
    LocalInterfaceAddr.s_addr = inet_addr("192.0.0.0");
    LocalInterfaceMask.s_addr = inet_addr("255.0.0.0");
    IrigMulticastGroup.s_addr = inet_addr("239.0.1.1");
    for (iInterfaceIdx = 0; iInterfaceIdx < iNumInterfaces; iInterfaceIdx++)
        {
        if ((NetInterfaces[iInterfaceIdx].s_addr & LocalInterfaceMask.s_addr) ==
LocalInterfaceAddr.s_addr)
            {
            join_source_group(m_suNetHandle[iHandle].suIrigSocket, IrigMulticastGroup,
NetInterfaces[iInterfaceIdx]);
            break;
            }
        }
#endif

    // Make sure the receive buffer is big enough for at least one UDP packet
    m_suNetHandle[iHandle].ulRcvBufferLen     = RCV_BUFFER_START_SIZE;
    m_suNetHandle[iHandle].pchRcvBuffer       = (char *)malloc(RCV_BUFFER_START_SIZE);

    m_suNetHandle[iHandle].ulRcvBufferDataLen = 0L;
    m_suNetHandle[iHandle].bBufferReady       = bFALSE;
    m_suNetHandle[iHandle].ulBufferPosIdx     = 0L;
    m_suNetHandle[iHandle].bGotFirstSegment   = bFALSE;

    m_suNetHandle[iHandle].enNetMode          = I106_READ_NET_STREAM;

    return I106_OK;
    }



/* ------------------------------------------------------------------------ */

/// Open an IRIG 106 Live Data Streaming send socket
EnI106Status I106_CALL_DECL
    enI106_OpenNetStreamWrite(int iHandle, uint32_t uIpAddress, uint16_t uUdpPort)
    {
    int                     iIdx;
    int                     iResult;
#ifdef SO_MAX_MSG_SIZE
```

```
    int                     iMaxMsgSizeLen;
#endif
#if defined(_MSC_VER)
    WORD                    wVersionRequested;
    WSADATA                 wsaData;
    DWORD                   iMaxMsgSize;
#else
    socklen_t               iMaxMsgSize;
#endif


    // Initialize handle data if necessary
    if (m_bHandlesInited == bFALSE)
        {
        for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
            {
            m_suNetHandle[iIdx].enNetMode  = I106_CLOSED;
            m_suNetHandle[iIdx].uUdpSeqNum = 0;
            }
        m_bHandlesInited = bTRUE;
        } // end if file handles not inited yet


#if defined(_MSC_VER)
    // Initialize WinSock, request version 2.2
    wVersionRequested = MAKEWORD(2, 2);
    iResult = WSAStartup(wVersionRequested, &wsaData);

    if (iResult != 0)
        {
//      printf("Unable to initialize Winsock 2.2\n");
        return I106_OPEN_ERROR;
        }
#endif

    // Create a socket for writing to UDP
    m_suNetHandle[iHandle].suIrigSocket = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    if (m_suNetHandle[iHandle].suIrigSocket == INVALID_SOCKET)
        {
//        printf("socket() failed with error: %ld\n", WSAGetLastError());
#if defined(_MSC_VER)
        WSACleanup();
#endif
        return I106_OPEN_ERROR;
        }

    // Fill in the remote host information
    m_suNetHandle[iHandle].suSendIpAddress.sin_family      = AF_INET;
    m_suNetHandle[iHandle].suSendIpAddress.sin_port        = htons(uUdpPort);
    m_suNetHandle[iHandle].suSendIpAddress.sin_addr.s_addr = htonl(uIpAddress);

#ifdef SO_MAX_MSG_SIZE
    // getsockopt to retrieve the value of option SO_MAX_MSG_SIZE after a socket has
been created.
    iMaxMsgSizeLen = sizeof(iMaxMsgSize);
    iResult = getsockopt(m_suNetHandle[iHandle].suIrigSocket, SOL_SOCKET,
SO_MAX_MSG_SIZE, (char *)&iMaxMsgSize, &iMaxMsgSizeLen);
#else
    iResult = 1;
#endif

    if (iResult == 0)
        {
```

```
        // Use smaller, taking into account Ch 10 UDP transfer header
        m_suNetHandle[iHandle].uMaxUdpSize = MIN(iMaxMsgSize-6,MAX_UDP_WRITE_SIZE);
        }
    else
        m_suNetHandle[iHandle].uMaxUdpSize = MAX_UDP_WRITE_SIZE;

    m_suNetHandle[iHandle].enNetMode = I106_WRITE_NET_STREAM;

    return I106_OK;
    }


/* ---------------------------------------------------------------------- */

EnI106Status I106_CALL_DECL
    enI106_CloseNetStream(int iHandle)
    {

#ifdef MULTICAST
    // Restore the appropriate interface out of multicast receive mode
    iNumInterfaces = GetInterfaces(NetInterfaces, 10);
    LocalInterfaceAddr.s_addr = inet_addr("192.0.0.0");
    LocalInterfaceMask.s_addr = inet_addr("255.0.0.0");
    IrigMulticastGroup.s_addr = inet_addr("224.0.0.1");
    for (iInterfaceIdx = 0; iInterfaceIdx < iNumInterfaces; iInterfaceIdx++)
        {
        if ((NetInterfaces[iInterfaceIdx].s_addr & LocalInterfaceMask.s_addr) ==
LocalInterfaceAddr.s_addr)
            {
            leave_source_group(IrigSocket, IrigMulticastGroup,
NetInterfaces[iInterfaceIdx]);
            break;
            }
        }
#endif

    switch (m_suNetHandle[iHandle].enNetMode)
        {
        case I106_READ_NET_STREAM :
            // Close the receive socket
#if defined(_MSC_VER)
            closesocket(m_suNetHandle[iHandle].suIrigSocket);
            WSACleanup();
#else
            close(m_suNetHandle[iHandle].suIrigSocket);
#endif
            // Free up allocated memory
            free(m_suNetHandle[iHandle].pchRcvBuffer);
            m_suNetHandle[iHandle].pchRcvBuffer       = NULL;
            m_suNetHandle[iHandle].ulRcvBufferLen     = 0L;
            break;

        case I106_WRITE_NET_STREAM :
            // Close the transmit socket
#if defined(_MSC_VER)
            closesocket(m_suNetHandle[iHandle].suIrigSocket);
            WSACleanup();
#else
            close(m_suNetHandle[iHandle].suIrigSocket);
#endif
            break;

        default :
```

```
            break;
        } // end switch on enNetMode

    // Mark this case closed
    m_suNetHandle[iHandle].enNetMode = I106_CLOSED;

    return I106_OK;
    }



// ----------------------------------------------------------------------
//! @brief Utility method for cross-platform recvmsg
//! @details Used by enI106_ReadNetStream to split a read across the
//!          header buffer and a body buffer.
//! @param suSocket The SOCKET handle to recv from
//! @param[out] pvBuffer1 Pointer to the first buffer to fill
//! @param[in]  ulBufLen1 Length of the first buffer, in bytes
//! @param[out] pvBuffer2 Pointer to the second buffer to fill
//! @param[in]  ulBufLen2 Length of the second buffer, in bytes
//! @param[out] ulBytesRcvdOut Returns the number of bytes received and copied into
the buffers
//! @return I106_OK On success;
//!         ulBytesRcvdOut contains the number of bytes read.
//! @return I106_MORE_DATA On a successful, but truncated, read;
//!         ulBytesRcvdOut contains the actual number of bytes read. Some data was
lost.
//! @return I106_READ_ERROR On error;
//!         ulBytesRcvdOut is undefined
static EnI106Status
    RecvMsgSplit(SOCKET suSocket,
                 void * const pvBuffer1,
                 unsigned long ulBufLen1,
                 void * const pvBuffer2,
                 unsigned long ulBufLen2,
                 unsigned long * pulBytesRcvdOut)
#if defined(_MSC_VER)
    {
    WSABUF         asuUdpRcvBuffs[2];
    DWORD          UdpRcvFlags = 0;
    DWORD          dwBytesRcvd = 0;
    int            iResult = 0;

    // Setup the message buffer structure
    asuUdpRcvBuffs[0].len  = ulBufLen1;
    asuUdpRcvBuffs[0].buf = (char *)pvBuffer1;
    asuUdpRcvBuffs[1].len  = ulBufLen2;
    asuUdpRcvBuffs[1].buf = (char *)pvBuffer2;

    iResult = WSARecv(suSocket, asuUdpRcvBuffs, 2, &dwBytesRcvd, &UdpRcvFlags, NULL,
NULL);
    if( pulBytesRcvdOut )
        *pulBytesRcvdOut = (unsigned long)dwBytesRcvd;

    if( 0 == iResult )
        return I106_OK;
    else
        {
        int const err = WSAGetLastError();
        if( WSAEMSGSIZE == err )
            return I106_MORE_DATA;
        else
            return I106_READ_ERROR;
```

```
        }
    }
#else
    {
    struct msghdr  suMsgHdr = { 0 };
    struct iovec   asuUdpRcvBuffs[2];
    const int      UdpRcvFlags = 0;
    ssize_t        iResult = 0;

    // Setup the message buffer structure
    suMsgHdr.msg_iov    = asuUdpRcvBuffs;
    suMsgHdr.msg_iovlen = 2;

    asuUdpRcvBuffs[0].iov_len  = ulBufLen1;
    asuUdpRcvBuffs[0].iov_base = (char *)pvBuffer1;
    asuUdpRcvBuffs[1].iov_len  = ulBufLen2;
    asuUdpRcvBuffs[1].iov_base = (char *)pvBuffer2;

    iResult = recvmsg(suSocket, &suMsgHdr, UdpRcvFlags);
    if( pulBytesRcvdOut )
        *pulBytesRcvdOut = (unsigned long)iResult;

    if (iResult < 0)
        return I106_READ_ERROR;
    else if( MSG_TRUNC == suMsgHdr.msg_flags )
        return I106_MORE_DATA;
    else
        return I106_OK;
    }
#endif

// ----------------------------------------------------------------------
//! @brief Utility method for dropping a peek'd bad packet
//! @details If enI106_ReadNetStream MSG_PEEK's a packet that is too small,
//!     then we have to remove it from the socket buffer before we can bail.
//!     Otherwise, the next MSG_PEEK will see the same bad packet.
static void
    DropBadPacket(int iHandle)
{
    char dummy;
    // We don't care about the return value, we're failing anyways.
    (void)recvfrom(m_suNetHandle[iHandle].suIrigSocket, &dummy, sizeof(dummy), 0, 0,
0);
}


// ----------------------------------------------------------------------
// Get the next header.

int I106_CALL_DECL
    enI106_ReadNetStream(int           iHandle,
                         void        * pvBuffer,
                         unsigned int  iBuffSize)
    {
    // The minimum packet size needed for a valid segmented message packet
    enum { MIN_SEG_LEN = sizeof(SuUDP_Transfer_Header_Seg)-1 +
sizeof(SuI106Ch10Header) };

    int                             iResult;
    SuUDP_Transfer_Header_Seg       suUdpSeg;  // Same prefix unsegmented msg

    unsigned long                   ulBytesRcvd;
    int                             iCopySize;
```

```
    SuI106Ch10Header             * psuHeader;

    // If we don't have a buffer ready to read from then read network packets
    if (m_suNetHandle[iHandle].bBufferReady == bFALSE)
        {
        // Get ready for a new buffer of data
        m_suNetHandle[iHandle].bBufferReady     = bFALSE;
        m_suNetHandle[iHandle].bGotFirstSegment = bFALSE;
        m_suNetHandle[iHandle].ulBufferPosIdx   = 0L;

        // Read until we've got a complete Ch 10 packet(s)
        while (m_suNetHandle[iHandle].bBufferReady == bFALSE)
            {
            // Peek at the message to determine the msg type
            // (segmented or non-segmented)
            iResult = recvfrom(m_suNetHandle[iHandle].suIrigSocket, (char *)&suUdpSeg,
sizeof(suUdpSeg), MSG_PEEK, NULL, NULL);
//printf("recvfrom = %d\n", iResult);
#if defined(_MSC_VER)
            // Make the WinSock return code more like POSIX to simplify the logic
            // WinSock returns -1 when the message is larger than the buffer
            // Thus, (iResult==-1) && WSAEMSGSIZE is expected, as we're only reading
            // the header
            if( (iResult == -1)  )
                {
                int const err = WSAGetLastError(); // called out for debugging
                if( err == WSAEMSGSIZE)
                    iResult = sizeof(suUdpSeg); // The buffer was filled
                }
#endif

            if( iResult == -1 )
                {
                enI106_DumpNetStream(iHandle);
                return -1;
                }

            // If I don't have at least enough for a common header then drop and bail
            // We'll check length again later, which depends on the msg type
            if( iResult < UDP_Transfer_Header_NonSeg_Len )
                {
//printf("msg bytes (%d) < transfer header length (%d)\n",
//iResult, UDP_Transfer_Header_NonSeg_Len);
                // Because we're peeking, we have to make sure to drop the bad packet.
                DropBadPacket(iHandle);
                enI106_DumpNetStream(iHandle);
                continue;
                }

            //! @todo Check the version field for a known version

            // Check and handle UDP sequence number
            if (suUdpSeg.uSeqNum != m_suNetHandle[iHandle].uUdpSeqNum+1)
                {
                enI106_DumpNetStream(iHandle);
//printf("UDP Sequence Gap - %u  %u\n",
//m_suNetHandle[iHandle].uUdpSeqNum, suUdpSeg.uSeqNum);
                }
            m_suNetHandle[iHandle].uUdpSeqNum = suUdpSeg.uSeqNum;

            // Handle full and segmented packet types
            switch (suUdpSeg.uMsgType)
```

```
                {
                case 0 : // Full packet(s)
//printf("Full - ");

                    iResult = RecvMsgSplit(m_suNetHandle[iHandle].suIrigSocket,
                                           &suUdpSeg,
                                           UDP_Transfer_Header_NonSeg_Len,
                                           m_suNetHandle[iHandle].pchRcvBuffer,
                                           m_suNetHandle[iHandle].ulRcvBufferLen,
                                           &ulBytesRcvd);
                    if (I106_OK != iResult)
                        {
                        enI106_DumpNetStream(iHandle);
                        if( I106_READ_ERROR == iResult )
                            return -1;
                        else
                            continue;
                        }

//printf("Size = %lu\n", ulBytesRcvd - UDP_Transfer_Header_NonSeg_Len);

                    m_suNetHandle[iHandle].ulRcvBufferDataLen = ulBytesRcvd -
UDP_Transfer_Header_NonSeg_Len;
                    m_suNetHandle[iHandle].bBufferReady       = bTRUE;
                    m_suNetHandle[iHandle].ulBufferPosIdx     = 0L;
                    break;

                case 1 : // Segmented packet
//printf("Segmented - ");

                    // We need at least enough to do the next read.
                    if( iResult < UDP_Transfer_Header_Seg_Len )
                        {
                        DropBadPacket(iHandle);
                        enI106_DumpNetStream(iHandle);
                        continue;
                        }

                    // Always write to the beginning of the buffer while waiting
                    // for the first segment.
                    // The first UDP packet is guaranteed to fit our default
                    // starting size
                    if (m_suNetHandle[iHandle].bGotFirstSegment == bFALSE)
                        {
                        iResult = RecvMsgSplit(m_suNetHandle[iHandle].suIrigSocket,
                                               &suUdpSeg,
                                               UDP_Transfer_Header_Seg_Len,
                                               m_suNetHandle[iHandle].pchRcvBuffer,
                                               m_suNetHandle[iHandle].ulRcvBufferLen,
                                               &ulBytesRcvd);
                        }
                    else
                        {
                        iResult = RecvMsgSplit(m_suNetHandle[iHandle].suIrigSocket,
                                               &suUdpSeg,
                                               UDP_Transfer_Header_Seg_Len,

&(m_suNetHandle[iHandle].pchRcvBuffer[suUdpSeg.uSegmentOffset]),
                                               m_suNetHandle[iHandle].ulRcvBufferLen -
suUdpSeg.uSegmentOffset,
                                               &ulBytesRcvd);
                        }
```

```
                    if (I106_OK != iResult)
                        {
                        enI106_DumpNetStream(iHandle);
                        if( I106_READ_ERROR == iResult )
                            return -1;
                        else
                            continue;
                        }

//printf("Offset = %u\n", suUdpSeg.uSegmentOffset);

                    // Make sure we can access Ch 10 header info
                    if( ulBytesRcvd < MIN_SEG_LEN )
                        {
                        DropBadPacket(iHandle);
                        enI106_DumpNetStream(iHandle);
                        continue;
                        }

                    psuHeader = (SuI106Ch10Header
*)m_suNetHandle[iHandle].pchRcvBuffer;

                    // If it's the first packet then figure out if our buffer is large
enough for the whole Ch10 packet
                    if (suUdpSeg.uSegmentOffset == 0)
                        {
                        if (psuHeader->ulPacketLen >
m_suNetHandle[iHandle].ulRcvBufferLen)
                            {
                            m_suNetHandle[iHandle].ulRcvBufferLen = psuHeader-
>ulPacketLen + 0x4000;
                            m_suNetHandle[iHandle].pchRcvBuffer   = (char
*)realloc(m_suNetHandle[iHandle].pchRcvBuffer,m_suNetHandle[iHandle].ulRcvBufferLen);
                            psuHeader = (SuI106Ch10Header
*)m_suNetHandle[iHandle].pchRcvBuffer;
                            } // end if buffer too small for whole Ch 10 packet
                        m_suNetHandle[iHandle].bGotFirstSegment   = bTRUE;
                        m_suNetHandle[iHandle].ulRcvBufferDataLen = psuHeader-
>ulPacketLen;
                        } // end if first packet

                    // If we've gotten the first and last packets then mark the buffer
as full and ready
                    if ((m_suNetHandle[iHandle].bGotFirstSegment == bTRUE) &&
// First UDP buffer
                        ((suUdpSeg.uSegmentOffset + ulBytesRcvd -
UDP_Transfer_Header_Seg_Len) >= psuHeader->ulPacketLen)) // Last UDP buffer
                        {
//if ((suUdpSeg.uSegmentOffset + ulBytesRcvd - UDP_Transfer_Header_Seg_Len) >
psuHeader->ulPacketLen)
    //printf("Last packet too long");
                        m_suNetHandle[iHandle].bBufferReady     = bTRUE;
                        m_suNetHandle[iHandle].bGotFirstSegment = bFALSE;
                        m_suNetHandle[iHandle].ulBufferPosIdx     = 0L;
                        } // end if got first and last packet

                    break;

                default :
                    // The peek'd packet specifies some unknown/junk message type
                    // Toss this packet so the MSG_PEEK doesn't loop on it endlessly
                    DropBadPacket(iHandle);
                    enI106_DumpNetStream(iHandle);
```

```
                    continue;
                } // end switch on UDP packet type
            } // end while reading for a complete buffer
        } // end if called and buffer not ready

    // Copy data to the user buffer
    iCopySize = MIN(m_suNetHandle[iHandle].ulRcvBufferDataLen -
m_suNetHandle[iHandle].ulBufferPosIdx, iBuffSize);
    memcpy(pvBuffer,
&m_suNetHandle[iHandle].pchRcvBuffer[m_suNetHandle[iHandle].ulBufferPosIdx],
iCopySize);

    // Update buffer status
    m_suNetHandle[iHandle].ulBufferPosIdx += iCopySize;
    if (m_suNetHandle[iHandle].ulBufferPosIdx >=
m_suNetHandle[iHandle].ulRcvBufferDataLen)
        {
        m_suNetHandle[iHandle].bBufferReady = bFALSE;
        }

    return iCopySize;
    }


// ---------------------------------------------------------------------------

// Manipulate receive buffer
// -------------------------

// Invalidate the receive buffer so that the next enI106_ReadNetStream()
// causes a new complete buffer to be read from the network

EnI106Status I106_CALL_DECL
    enI106_DumpNetStream(int iHandle)
    {
    m_suNetHandle[iHandle].bBufferReady     = bFALSE;
    m_suNetHandle[iHandle].bGotFirstSegment = bFALSE;
    m_suNetHandle[iHandle].ulBufferPosIdx   = 0L;

    return I106_OK;
    }



// ---------------------------------------------------------------------------

EnI106Status I106_CALL_DECL
    enI106_MoveReadPointer(int iHandle, long iRelOffset)
    {
    long    lNewPosition;

    lNewPosition = m_suNetHandle[iHandle].ulBufferPosIdx + iRelOffset;
    if (lNewPosition < 0)
        m_suNetHandle[iHandle].ulBufferPosIdx = 0L;

    else if ((unsigned long)lNewPosition >= m_suNetHandle[iHandle].ulRcvBufferDataLen)
        {
        m_suNetHandle[iHandle].ulBufferPosIdx = 0L;
        m_suNetHandle[iHandle].bBufferReady   = bFALSE;
        }

    else
        m_suNetHandle[iHandle].ulBufferPosIdx = (unsigned long)lNewPosition;
```

```
    return I106_OK;
    }


// ----------------------------------------------------------------------

EnI106Status I106_CALL_DECL
    enI106_WriteNetStream(int           iHandle,
                          void        * pvBuffer,
                          uint32_t      uBuffSize)
    {
    EnI106Status        enStatus;
    EnI106Status        enReturnStatus;
    void              * pvCurrSendBuffPos;
    uint32_t            uCurrSendBuffLen;
    SuI106Ch10Header  * psuCurrCh10Header;
    SuI106Ch10Header  * psuNextCh10Header;

    enReturnStatus = I106_OK;

    // Check for initialized
    if (m_suNetHandle[iHandle].enNetMode != I106_WRITE_NET_STREAM)
        return I106_NOT_OPEN;

    // THIS WOULD BE A GOOD PLACE TO CHECK DATA PACKET INTEGRITY SOMEDAY

    // Queue up the first IRIG packet
    psuCurrCh10Header = (SuI106Ch10Header *)pvBuffer;
    uCurrSendBuffLen  = psuCurrCh10Header->ulPacketLen;
    pvCurrSendBuffPos = pvBuffer;
//    psuNextCh10Header = (SuI106Ch10Header  *)((char *)pvBuffer + psuCurrCh10Header-
>ulPacketLen);

//    if ((char *)psuNextCh10Header >= ((char *)pvBuffer + uBuffSize))
//        psuNextCh10Header = NULL;

    // If psuNextCh10Header > pvBuffer + uBuffSize then this is a malformed buffer.
//    assert((char *)psuNextCh10Header <= ((char *)pvBuffer + uBuffSize));

    // Step through IRIG packets until the length would exceed max UDP packet size
    while (1==1)
        {
        // If current packet size > max then send segmented packet
        if (psuCurrCh10Header->ulPacketLen > m_suNetHandle[iHandle].uMaxUdpSize)
            {
            // This big packet had better be the first one in our current send buffer
//          assert(pvCurrSendBuffPos == psuCurrCh10Header);
            assert(psuCurrCh10Header->uSync == 0xEB25);

            // Send segmented packet
            enStatus = enI106_WriteNetSegmented(iHandle, pvCurrSendBuffPos,
uCurrSendBuffLen);
            if (enStatus != I106_OK)
                enReturnStatus = enStatus;

            // Update pointer to the next IRIG packet, or exit if we are done
            pvCurrSendBuffPos = ((char *)pvCurrSendBuffPos + psuCurrCh10Header-
>ulPacketLen);
            psuCurrCh10Header = (SuI106Ch10Header *)pvCurrSendBuffPos;
            if ((char *)pvCurrSendBuffPos >= ((char *)pvBuffer + uBuffSize))
                {
```

```
                break;
                }
            else
                {
                uCurrSendBuffLen  = psuCurrCh10Header->ulPacketLen;
                continue;
                }
            } // end if big segmented packet

        // If no more Ch 10 packets then send what we have, done
        if (((char *)psuCurrCh10Header + psuCurrCh10Header->ulPacketLen) >= ((char
*)pvBuffer + uBuffSize))
            {
            // If psuNextCh10Header > pvBuffer + uBuffSize then this is a malformed
buffer.
//          assert((char *)psuNextCh10Header <= ((char *)pvBuffer + uBuffSize));

            assert(uCurrSendBuffLen <= m_suNetHandle[iHandle].uMaxUdpSize);
            // Send non-segmented packet
            enStatus = enI106_WriteNetNonSegmented(iHandle, pvCurrSendBuffPos,
uCurrSendBuffLen);
            if (enStatus != I106_OK)
                enReturnStatus = enStatus;

            // Done sending packets
            break;
            } // end if last message(s) in buffer

        // There is another buffer so let's check its size. If next packet would put
us over
        // max size then send what we have
        psuNextCh10Header = (SuI106Ch10Header *)((char *)psuCurrCh10Header +
psuCurrCh10Header->ulPacketLen);
        // Might want to validate sync word, packet header checksum, and packet
checksum
        assert(psuNextCh10Header->uSync == 0xEB25);

        if ((uCurrSendBuffLen + psuNextCh10Header->ulPacketLen) >
m_suNetHandle[iHandle].uMaxUdpSize)
            {
            assert(psuCurrCh10Header->uSync == 0xEB25);

            // Send non-segmented packet
            enStatus = enI106_WriteNetNonSegmented(iHandle, pvCurrSendBuffPos,
uCurrSendBuffLen);
            if (enStatus != I106_OK)
                enReturnStatus = enStatus;

            // Update pointer to the next IRIG packet, or exit if we are done
            pvCurrSendBuffPos = psuNextCh10Header;
            psuCurrCh10Header = psuNextCh10Header;
            uCurrSendBuffLen  = psuCurrCh10Header->ulPacketLen;
            if ((char *)pvCurrSendBuffPos >= ((char *)pvBuffer + uBuffSize))
                {
                // We should never get here but just in case
                assert(1==0);
                break;
                }
            else
                {
                continue;
                }
            } // end if next packet puts us over the max size
```

```
        // Nothing to send so advance to the next Ch 10 packet
        psuCurrCh10Header += psuCurrCh10Header->ulPacketLen;
        // Might want to validate sync word, packet header checksum, and packet
checksum
        assert(psuCurrCh10Header->uSync == 0xEB25);

        } // Done stepping through all IRIG packets

    return enReturnStatus;
    }


// ----------------------------------------------------------------------

// Send a non-segmented UDP packet

EnI106Status I106_CALL_DECL
    enI106_WriteNetNonSegmented(
            int           iHandle,
            void        * pvBuffer,
            uint32_t      uBuffSize)
    {
    EnI106Status        enReturnStatus;

#if defined(_MSC_VER)
//  SOCKET_ADDRESS      suMsSendIpAddress;
    WSAMSG              suMsMsgInfo;
    WSABUF              suMsBuffInfo[2];
    WSABUF              suMsControl;
    DWORD               lBytesSent;
#endif
    int                 iSendStatus;

    SuUDP_Transfer_Header_NonSeg    suUdpHeaderNonSeg;

    enReturnStatus = I106_OK;

    // Setup the non-segemented transfer header
    suUdpHeaderNonSeg.uVersion = 1;
    suUdpHeaderNonSeg.uMsgType = 0;
    suUdpHeaderNonSeg.uSeqNum  = m_suNetHandle[iHandle].uUdpSeqNum;

    // Send the IRIG UDP packet
#if defined(_MSC_VER)
    // I don't really want or need control data. I hope this doesn't
    // cause WSASendMsg() to fail.
    suMsControl.buf           = NULL;
    suMsControl.len           = 0;

    // Setup pointers to the data to be sent.
    suMsBuffInfo[0].buf       = (CHAR *)&suUdpHeaderNonSeg;
    suMsBuffInfo[0].len       = 4;
    suMsBuffInfo[1].buf       = (CHAR *)pvBuffer;
    suMsBuffInfo[1].len       = uBuffSize;

    // Setup the send info for WSASendMsg()
    suMsMsgInfo.name          = (SOCKADDR*)&(m_suNetHandle[iHandle].suSendIpAddress);
// THIS IS AMBIGUOUS IN MSDN
    suMsMsgInfo.namelen       = sizeof(m_suNetHandle[iHandle].suSendIpAddress);
    suMsMsgInfo.lpBuffers     = suMsBuffInfo;
    suMsMsgInfo.dwBufferCount = 2;
    suMsMsgInfo.Control       = suMsControl;
```

```
    suMsMsgInfo.dwFlags       = 0;

    // Send it. Done!
    iSendStatus = WSASendMsg(m_suNetHandle[iHandle].suIrigSocket, &suMsMsgInfo, 0,
&lBytesSent, NULL, NULL);
    if (iSendStatus != 0)
        enReturnStatus = I106_WRITE_ERROR;
#else
// TODO - LINUX CODE
#endif

    // Increment the sequence number for next time
    m_suNetHandle[iHandle].uUdpSeqNum++;

    return enReturnStatus;
    }


// ----------------------------------------------------------------------------

// Send a segmented UDP packet

EnI106Status I106_CALL_DECL
    enI106_WriteNetSegmented(
            int           iHandle,
            void        * pvBuffer,
            uint32_t      uBuffSize)
    {
    EnI106Status        enReturnStatus;
    uint32_t            uBuffIdx;
    char              * pchBuffer;
    uint32_t            uSendSize;
    int                 iSendStatus;
    SuI106Ch10Header  * psuHeader;

#if defined(_MSC_VER)
    WSAMSG              suMsMsgInfo;
    WSABUF              suMsBuffInfo[2];
    WSABUF              suMsControl;
    DWORD               lBytesSent;
#endif

    SuUDP_Transfer_Header_Seg       suUdpHeaderSeg;

    enReturnStatus = I106_OK;

    // Setup the segemented transfer header
    psuHeader = (SuI106Ch10Header *)pvBuffer;

    memset(&suUdpHeaderSeg, 0, 12);
    suUdpHeaderSeg.uVersion     = 1;
    suUdpHeaderSeg.uMsgType     = 1;
    suUdpHeaderSeg.uChID        = psuHeader->uChID;
    suUdpHeaderSeg.uChanSeqNum  = psuHeader->ubySeqNum;

    // Send the IRIG UDP packets
    uBuffIdx = 0;
    while (uBuffIdx < uBuffSize)
        {
        suUdpHeaderSeg.uSeqNum        = m_suNetHandle[iHandle].uUdpSeqNum;
        suUdpHeaderSeg.uSegmentOffset = uBuffIdx;

        pchBuffer  = (char *)pvBuffer + uBuffIdx;
```

```
        uSendSize = MIN(m_suNetHandle[iHandle].uMaxUdpSize, uBuffSize-uBuffIdx);
#if defined(_MSC_VER)
        // I don't really want or need control data. I hope this doesn't
        // cause WSASendMsg() to fail.
        suMsControl.buf           = NULL;
        suMsControl.len           = 0;

        // Setup pointers to the data to be sent.
        suMsBuffInfo[0].buf       = (CHAR *)&suUdpHeaderSeg;
        suMsBuffInfo[0].len       = 12;
        suMsBuffInfo[1].buf       = pchBuffer;
        suMsBuffInfo[1].len       = uSendSize;

        // Setup the send info for WSASendMsg()
        suMsMsgInfo.name          =
(SOCKADDR*)&(m_suNetHandle[iHandle].suSendIpAddress);  // THIS IS AMBIGUOUS IN MSDN
        suMsMsgInfo.namelen       = sizeof(m_suNetHandle[iHandle].suSendIpAddress);
        suMsMsgInfo.lpBuffers     = suMsBuffInfo;
        suMsMsgInfo.dwBufferCount = 2;
        suMsMsgInfo.Control       = suMsControl;
        suMsMsgInfo.dwFlags       = 0;

        // Send it. Done!
        iSendStatus = WSASendMsg(m_suNetHandle[iHandle].suIrigSocket, &suMsMsgInfo, 0,
&lBytesSent, NULL, NULL);
        if (iSendStatus != 0)
            {
            enReturnStatus = I106_WRITE_ERROR;
            break;
            }

#else
// TODO - LINUX CODE
#endif

        // Update the buffer index
        uBuffIdx += uSendSize;

        // Increment the sequence number for next time
        m_suNetHandle[iHandle].uUdpSeqNum++;

        } // end while not at the end of the buffer

    return enReturnStatus;
    }
#ifdef __cplusplus
} // end namespace
#endif
```

Appendix A-9. config.h

```
/****************************************************************************

 config.h - Define features and OS portability macros

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 ****************************************************************************/

#ifndef _config_h_
#define _config_h_

#ifdef __cplusplus
extern "C" {
#endif

// For best portability, time_t is assumed to be a 32 bit value.
// This needs to be set in the project properties. This forces
// a puke if it isn't set.
#if (_MSC_VER >= 1400) && !defined(_WIN64)
  #if !defined(_USE_32BIT_TIME_T)
  #pragma message("WARNING - '_USE_32BIT_TIME_T' not set!")
  #endif
#endif

/* The POSIX caseless string compare is strcasecmp(). MSVC uses the
 * non-standard stricmp(). Fix it up with a macro if necessary
 */

#if defined(_MSC_VER)
#define strcasecmp(s1, s2)          _stricmp(s1, s2)
#define strncasecmp(s1, s2, n)      _strnicmp(s1, s2, n)
#pragma warning(disable : 4996)
#endif
```

```
#define I106_CALL_DECL

// Turn on network support
// #define IRIG_NETWORKING

#ifdef __cplusplus
}
#endif

#endif
```

Appendix C

Example Program – Calculate Histogram

The following software program opens a Chapter 10 file for reading, calculates a running count of each packet type in each channel, and then prints out these totals. It demonstrates reading individual Chapter 10 packets, and parsing them based on packet type.

```
/*===========================================================================

 i106stat - Generate histogram-like statistics on a Irig 106 data file

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

 Created by Bob Baggerman

 ***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>

#include "config.h"
#include "stdint.h"
#include "irig106ch10.h"
#include "i106_time.h"

#include "i106_decode_time.h"
#include "i106_decode_1553f1.h"
```

```
#include "i106_decode_arinc429.h"
#include "i106_decode_tmats.h"


/*
 * Macros and definitions
 * ----------------------
 */

#define MAJOR_VERSION  "01"
#define MINOR_VERSION  "04"

#if !defined(bTRUE)
#define bTRUE   (1==1)
#define bFALSE  (1==0)
#endif

/*
 * Data structures
 * ---------------
 */


/* These hold the number of messages of each type for the histogram. */

// 1553 channel counts
typedef struct
    {
    unsigned long       ulTotalIrigPackets;
    unsigned long       ulTotalBusMsgs;
    unsigned long       ulTotalIrigPacketErrors;
    unsigned long       aulMsgs[0x4000];
    unsigned long       aulErrs[0x4000];
    unsigned long       ulErr1553Timeout;
    int                 bRT2RTFound;
    } SuChanInfo1553;

// ARINC 429 counts
typedef struct
    {
    unsigned long   aulMsgs[0x100][0x100];
    } SuARINC429;


// Per channel statistics
typedef struct
  {
    unsigned int        iChanID;
    int                 iPrevSeqNum;
    unsigned long       ulSeqNumError;
    unsigned char       szChanType[32];
    unsigned char       szChanName[32];
    SuChanInfo1553    * psu1553Info;
    unsigned long       ulUserDefined;
    unsigned long       ulIrigTime;
    unsigned long       ulAnalog;
    unsigned long       ulTMATS;
    unsigned long       ulEvents;
    unsigned long       ulIndex;
    unsigned long       ulPCM;
    unsigned long       ulMPEG2;
    unsigned long       ulUART;
    unsigned long       ulEthernet;
```

```
    SuARINC429        * paARINC429;
    unsigned long       ul16PP194;
    unsigned long       ulDiscrete;
    unsigned long       ulParallel;
    unsigned long       ulMessage;
    unsigned long       ulImage;
    unsigned long       ulTSPI;
    unsigned long       ulCAN;
    unsigned long       ulFibreChan;
    unsigned long       ulARINC664;
    unsigned long       ulOther;
    } SuChanInfo;


/*
 * Module data
 * -----------
 */

int                 m_bLogRT2RT;
int                 m_bVerbose;
unsigned char       m_aArincLabelMap[0x100];


/*
 * Function prototypes
 * -------------------
 */

void     vPrintCounts(SuChanInfo * psuChanInfo, FILE * psuOutFile);
void     vPrintTmats(SuTmatsInfo * psuTmatsInfo, FILE * psuOutFile);
void     vProcessTmats(SuTmatsInfo * psuTmatsInfo, SuChanInfo * apsuChanInfo[]);
void     vMakeArincLabelMap(unsigned char m_aArincLabelMap[]);
void     vUsage(void);


/* ---------------------------------------------------------------------- */

int main(int argc, char ** argv)
    {

    // Array of pointers to the SuChanInfo structure
    static SuChanInfo     * apsuChanInfo[0x10000];

    unsigned char           abyFileStartTime[6];
    unsigned char           abyStartTime[6];
    unsigned char           abyStopTime[6];
    int                     bFoundFileStartTime = bFALSE;
    int                     bFoundDataStartTime = bFALSE;
    unsigned long           ulReadErrors;
    unsigned long           ulBadPackets;
    unsigned long           ulTotal;

    FILE                  * psuOutFile;        // Output file handle
    int                     hI106In;
    char                    szInFile[255];     // Input file name
    char                    szOutFile[255];    // Output file name
    int                     iArgIdx;
    unsigned short          usPackedIdx;
    unsigned long           ulBuffSize = 0L;
    unsigned long           ulReadSize;

    unsigned int            uChanIdx;
```

```
    EnI106Status            enStatus;
    SuI106Ch10Header        suI106Hdr;
    Su1553F1_CurrMsg        su1553Msg;
    SuArinc429F0_CurrMsg    suArincMsg;
    SuTmatsInfo             suTmatsInfo;
    SuIrig106Time           suIrigTime;
    struct tm             * psuTmTime;
    char                    szTime[50];
    char                  * szDateTimeFmt = "%m/%d/%Y %H:%M:%S";
    char                  * szDayTimeFmt  = "%j:%H:%M:%S";
    char                  * szTimeFmt;

    unsigned char         * pvBuff = NULL;


// Make sure things stay on UTC

    putenv("TZ=GMT0");
    tzset();

/*
 * Initialize the channel info array pointers to all NULL
 */

    memset(apsuChanInfo, 0, sizeof(apsuChanInfo));
    ulTotal      = 0L;
    ulReadErrors = 0L;
    ulBadPackets = 0L;

/*
 * Process the command line arguements
 */

    if (argc < 2)
        {
        vUsage();
        return 1;
        }

    m_bVerbose    = bFALSE;               // No verbosity
    m_bLogRT2RT   = bFALSE;               // Don't keep track of RT to RT
    szInFile[0] = '\0';
    strcpy(szOutFile,"");                     // Default is stdout

    for (iArgIdx=1; iArgIdx<argc; iArgIdx++)
        {

        switch (argv[iArgIdx][0])
            {

            // Handle command line flags
            case '-' :
                switch (argv[iArgIdx][1])
                    {

                    case 'r' :                    // Log RT to RT
                        m_bLogRT2RT = bTRUE;
                        break;

                    case 'v' :                    // Verbose switch
                        m_bVerbose = bTRUE;
                        break;
```

```
                    default :
                        break;
                    } /* end flag switch */
                break;

            // Anything else must be a file name
            default :
                if (szInFile[0] == '\0') strcpy(szInFile, argv[iArgIdx]);
                else                     strcpy(szOutFile,argv[iArgIdx]);
                break;

            } /* end command line arg switch */
        } /* end for all arguments */

    if (strlen(szInFile)==0)
        {
        vUsage();
        return 1;
        }

/*
 * Opening banner
 * --------------
 */

    fprintf(stderr, "\nI106STAT "MAJOR_VERSION"."MINOR_VERSION"\n");
    fprintf(stderr, "Freeware Copyright (C) 2006 Irig106.org\n\n");

/*
 * Opens file and get everything init'ed
 * -------------------------------------
 */


    // Open file and allocate a buffer for reading data.
    enStatus = enI106Ch10Open(&hI106In, szInFile, I106_READ);
    switch (enStatus)
        {
        case I106_OPEN_WARNING :
            fprintf(stderr, "Warning opening data file : Status = %d\n", enStatus);
            break;
        case I106_OK :
            break;
        default :
            fprintf(stderr, "Error opening data file : Status = %d\n", enStatus);
            return 1;
            break;
        }

    enStatus = enI106_SyncTime(hI106In, bFALSE, 0);
    if (enStatus != I106_OK)
        {
        fprintf(stderr, "Error establishing time sync : Status = %d\n", enStatus);
        return 1;
        }

    // If output file specified then open it
    if (strlen(szOutFile) != 0)
        {
        psuOutFile = fopen(szOutFile,"w");
        if (psuOutFile == NULL)
            {
```

```
            fprintf(stderr, "Error opening output file\n");
            return 1;
            }
        }

    // No output file name so use stdout
    else
        {
        psuOutFile = stdout;
        }

    // Make the ARINC label map just in case
    vMakeArincLabelMap(m_aArincLabelMap);

    fprintf(stderr, "Computing histogram...\n");


/*
 * Loop until there are no more message whilst keeping track of all the
 * various message counts.
 * ----------------------------------------------------------------------
 */

    while (1==1)
        {

        // Read the next header
        enStatus = enI106Ch10ReadNextHeader(hI106In, &suI106Hdr);

        // Setup a one time loop to make it easy to break out on error
        do
            {
            if (enStatus == I106_EOF)
                {
                break;
                }

            // Check for header read errors
            if (enStatus != I106_OK)
                {
                ulReadErrors++;
                break;
                }

            // Make sure our buffer is big enough, size *does* matter
            if (ulBuffSize < uGetDataLen(&suI106Hdr))
                {
                pvBuff = realloc(pvBuff, uGetDataLen(&suI106Hdr));
                ulBuffSize = uGetDataLen(&suI106Hdr);
                }

            // Read the data buffer
            ulReadSize = ulBuffSize;
            enStatus = enI106Ch10ReadData(hI106In, ulBuffSize, pvBuff);

            // Check for data read errors
            if (enStatus != I106_OK)
                {
                ulReadErrors++;
                break;
                }

            // If this is a new channel, malloc some memory for counts and
```

```
            // set the pointer in the channel info array to it.
            if (apsuChanInfo[suI106Hdr.uChID] == NULL)
                {
                apsuChanInfo[suI106Hdr.uChID] = (SuChanInfo
*)malloc(sizeof(SuChanInfo));
                memset(apsuChanInfo[suI106Hdr.uChID], 0, sizeof(SuChanInfo));
                apsuChanInfo[suI106Hdr.uChID]->iChanID = suI106Hdr.uChID;
                // Now save channel type and name
                if (suI106Hdr.uChID == 0)
                    {
                    strcpy(apsuChanInfo[suI106Hdr.uChID]->szChanType, "RESERVED");
                    strcpy(apsuChanInfo[suI106Hdr.uChID]->szChanName, "SYSTEM");
                    }
                else
                    {
                    strcpy(apsuChanInfo[suI106Hdr.uChID]->szChanType, "UNKNOWN");
                    strcpy(apsuChanInfo[suI106Hdr.uChID]->szChanName, "UNKNOWN");
                    }
                }

            ulTotal++;
            if (m_bVerbose)
                fprintf(stderr, "%8.8ld Messages \r",ulTotal);

            // Save data start and stop times
            if ((suI106Hdr.ubyDataType != I106CH10_DTYPE_TMATS          ) &&
                (suI106Hdr.ubyDataType != I106CH10_DTYPE_IRIG_TIME      ) &&
                (suI106Hdr.ubyDataType != I106CH10_DTYPE_RECORDING_INDEX))
                {
                if (bFoundDataStartTime == bFALSE)
                    {
                    memcpy((char *)abyStartTime, (char *)suI106Hdr.aubyRefTime, 6);
                    bFoundDataStartTime = bTRUE;
                    }
                else
                    {
                    memcpy((char *)abyStopTime, (char *)suI106Hdr.aubyRefTime, 6);
                    }
                } // end if data message

            // Log the various data types
            switch (suI106Hdr.ubyDataType)
                {

                case I106CH10_DTYPE_USER_DEFINED :      // 0x00
                    apsuChanInfo[suI106Hdr.uChID]->ulUserDefined++;
                    break;

                case I106CH10_DTYPE_TMATS :             // 0x01
                    apsuChanInfo[suI106Hdr.uChID]->ulTMATS++;

                    // Only decode the first TMATS record
                    if (apsuChanInfo[suI106Hdr.uChID]->ulTMATS != 0)
                        {
                        // Save file start time
                        memcpy((char *)&abyFileStartTime, (char
*)suI106Hdr.aubyRefTime, 6);

                        // Process TMATS info for later use
                        memset( &suTmatsInfo, 0, sizeof(suTmatsInfo) );
                        enI106_Decode_Tmats(&suI106Hdr, pvBuff, &suTmatsInfo);
                        if (enStatus != I106_OK)
                            break;
```

```
                        vProcessTmats(&suTmatsInfo, apsuChanInfo);
                        }
                    break;

                case I106CH10_DTYPE_RECORDING_EVENT :   // 0x02
                    apsuChanInfo[suI106Hdr.uChID]->ulEvents++;
                    break;

                case I106CH10_DTYPE_RECORDING_INDEX :   // 0x03
                    apsuChanInfo[suI106Hdr.uChID]->ulIndex++;
                    break;

                case I106CH10_DTYPE_PCM_FMT_0 :         // 0x08
                case I106CH10_DTYPE_PCM_FMT_1 :         // 0x09
                    apsuChanInfo[suI106Hdr.uChID]->ulPCM++;
                    break;

                case I106CH10_DTYPE_IRIG_TIME :         // 0x11
                    apsuChanInfo[suI106Hdr.uChID]->ulIrigTime++;
                    break;

                case I106CH10_DTYPE_1553_FMT_1 :        // 0x19

                    // If first 1553 message for this channel, setup the 1553 counts
                    if (apsuChanInfo[suI106Hdr.uChID]->psu1553Info == NULL)
                        {
                        apsuChanInfo[suI106Hdr.uChID]->psu1553Info =
                            malloc(sizeof(SuChanInfo1553));
                        memset(apsuChanInfo[suI106Hdr.uChID]->psu1553Info, 0x00,
sizeof(SuChanInfo1553));
                        }

                    apsuChanInfo[suI106Hdr.uChID]->psu1553Info->ulTotalIrigPackets++;

                    // Step through all 1553 messages
                    enStatus = enI106_Decode_First1553F1(&suI106Hdr, pvBuff,
&su1553Msg);
                    if (enStatus == I106_OK)
                        {
                        while (enStatus == I106_OK)
                            {
                            // Update message count
                            apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>ulTotalBusMsgs++;
                            usPackedIdx = (su1553Msg.psuCmdWord1->uValue >> 5) &
0x3FFF;
                            apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>aulMsgs[usPackedIdx]++;

                            // Update the error counts
                            if (su1553Msg.psu1553Hdr->bMsgError != 0)
                                apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>aulErrs[usPackedIdx]++;

                            if (su1553Msg.psu1553Hdr->bRespTimeout != 0)
                                apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>ulErr1553Timeout++;

                            // If logging RT to RT then do it for second command word
                            if (su1553Msg.psu1553Hdr->bRT2RT == 1)
                                {
                                apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>bRT2RTFound = bTRUE;
```

```
                                if (m_bLogRT2RT==bTRUE)
                                    {
                                    usPackedIdx = (su1553Msg.psuCmdWord2->uValue >> 5)
& 0x3FFF;
                                    apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>aulMsgs[usPackedIdx]++;
                                    } // end if logging RT to RT
                                } // end if RT to RT

                            // Get the next 1553 message
                            enStatus = enI106_Decode_Next1553F1(&su1553Msg);
                            } // end while I106_OK
                        } // end if Decode First 1553 OK

                    // Decode not good so mark it as a packet error
                    else
                        {
                        apsuChanInfo[suI106Hdr.uChID]->psu1553Info-
>ulTotalIrigPacketErrors++;
                        }

                    break;

                case I106CH10_DTYPE_ANALOG :            // 0x21
                    apsuChanInfo[suI106Hdr.uChID]->ulAnalog++;
                    break;

                case I106CH10_DTYPE_ARINC_429_FMT_0 :   // 0x38
                    // If first ARINC 429 message for this channel, setup the counts
                    if (apsuChanInfo[suI106Hdr.uChID]->paARINC429 == NULL)
                        {
                        apsuChanInfo[suI106Hdr.uChID]->paARINC429 =
                            malloc(sizeof(SuARINC429));
                        memset(apsuChanInfo[suI106Hdr.uChID]->paARINC429, 0x00,
sizeof(SuARINC429));
                        }

                    // Step through all ARINC 429 messages
                    enStatus = enI106_Decode_FirstArinc429F0(&suI106Hdr, pvBuff,
&suArincMsg);
                    if (enStatus == I106_OK)
                        {
                        while (enStatus == I106_OK)
                            {
                            unsigned char   uBus;
                            unsigned char   uLabel;

                            // Update message count
                            uBus   = (unsigned char)suArincMsg.psu429Hdr->uBusNum;
                            uLabel = (unsigned
char)m_aArincLabelMap[suArincMsg.psu429Data->uLabel];
                            apsuChanInfo[suI106Hdr.uChID]->paARINC429-
>aulMsgs[uBus][uLabel]++;

                            // Get the next ARINC 429 message
                            enStatus = enI106_Decode_NextArinc429F0(&suArincMsg);
                            } // end while I106_OK
                        } // end if Decode First ARINC 429 OK

                    break;

                case I106CH10_DTYPE_VIDEO_FMT_0 :       // 0x40
```

```
    case I106CH10_DTYPE_VIDEO_FMT_1 :       // 0x41
    case I106CH10_DTYPE_VIDEO_FMT_2 :       // 0x42
    case I106CH10_DTYPE_VIDEO_FMT_3 :       // 0x43
    case I106CH10_DTYPE_VIDEO_FMT_4 :       // 0x44
        apsuChanInfo[suI106Hdr.uChID]->ulMPEG2++;
        break;

    case I106CH10_DTYPE_UART_FMT_0 :        // 0x50
        apsuChanInfo[suI106Hdr.uChID]->ulUART++;
        break;

    case I106CH10_DTYPE_ETHERNET_FMT_0 :    // 0x68
        apsuChanInfo[suI106Hdr.uChID]->ulEthernet++;
        break;

    case I106CH10_DTYPE_16PP194 :           // 0x1A
        apsuChanInfo[suI106Hdr.uChID]->ul16PP194++;
        break;

    case I106CH10_DTYPE_DISCRETE :          // 0x29
        apsuChanInfo[suI106Hdr.uChID]->ulDiscrete++;
        break;

    case I106CH10_DTYPE_PARALLEL_FMT_0 :    // 0x60
        apsuChanInfo[suI106Hdr.uChID]->ulParallel++;
        break;

    case I106CH10_DTYPE_MESSAGE :           // 0x30
        apsuChanInfo[suI106Hdr.uChID]->ulMessage++;
        break;

    case I106CH10_DTYPE_IMAGE_FMT_1 :       // 0x48
    case I106CH10_DTYPE_IMAGE_FMT_2 :       // 0x49
        apsuChanInfo[suI106Hdr.uChID]->ulImage++;
        break;

    case I106CH10_DTYPE_TSPI_FMT_0 :        // 0x70
    case I106CH10_DTYPE_TSPI_FMT_1 :        // 0x71
    case I106CH10_DTYPE_TSPI_FMT_2 :        // 0x72
        apsuChanInfo[suI106Hdr.uChID]->ulTSPI++;
        break;

    case I106CH10_DTYPE_CAN :               // 0x78
        apsuChanInfo[suI106Hdr.uChID]->ulCAN++;
        break;

    case I106CH10_DTYPE_FC_FMT_0 :          // 0x79
    case I106CH10_DTYPE_FC_FMT_1 :          // 0x7A
        apsuChanInfo[suI106Hdr.uChID]->ulFibreChan++;
        break;

    case I106CH10_DTYPE_ETHERNET_A664 :     // 0x69
        apsuChanInfo[suI106Hdr.uChID]->ulARINC664++;
        break;

    default:
        apsuChanInfo[suI106Hdr.uChID]->ulOther++;
        break;

    } // end switch on message type

} while (bFALSE); // end one time loop
```

```
        // If EOF break out of main read loop
        if (enStatus == I106_EOF)
            {
            break;
            }

        }   /* End while */


/*
 * Now print out the results of histogram.
 * ---------------------------------------
 */

//    vPrintTmats(&suTmatsInfo, psuOutFile);

    fprintf(psuOutFile,"\n=-=-= Message Totals by Channel and Type =-=-=\n\n");
    for (uChanIdx=0; uChanIdx<0x1000; uChanIdx++)
        {
        if (apsuChanInfo[uChanIdx] != NULL)
            {
            vPrintCounts(apsuChanInfo[uChanIdx], psuOutFile);
            }
        }


    fprintf(psuOutFile,"=-=-= File Time Summary =-=-=\n\n");

    enI106_Rel2IrigTime(hI106In, abyFileStartTime, &suIrigTime);
    if (suIrigTime.enFmt == I106_DATEFMT_DMY)
        szTimeFmt = szDateTimeFmt;
    else
        szTimeFmt = szDayTimeFmt;
    psuTmTime = gmtime((time_t *)&(suIrigTime.ulSecs));
    strftime(szTime, 50, szTimeFmt, psuTmTime);
    fprintf(psuOutFile,"File Start %s\n",  szTime);

    enI106_Rel2IrigTime(hI106In, abyStartTime, &suIrigTime);
    psuTmTime = gmtime((time_t *)&(suIrigTime.ulSecs));
    strftime(szTime, 50, szTimeFmt, psuTmTime);
    fprintf(psuOutFile,"Data Start %s\n",  szTime);

    enI106_Rel2IrigTime(hI106In, abyStopTime, &suIrigTime);
    psuTmTime = gmtime((time_t *)&(suIrigTime.ulSecs));
    strftime(szTime, 50, szTimeFmt, psuTmTime);
    fprintf(psuOutFile,"Data Stop  %s\n\n",  szTime);

    fprintf(psuOutFile,"\nTOTAL PACKETS:    %10lu\n\n", ulTotal);

/*
 *  Free dynamic memory.
 */

    free(pvBuff);
    pvBuff = NULL;

    fclose(psuOutFile);

    return 0;
    }
```

```
/* ---------------------------------------------------------------------- */

void vPrintCounts(SuChanInfo * psuChanInfo, FILE * psuOutFile)
    {
    long            lMsgIdx;

    // Make Channel ID line lead-in string
    fprintf(psuOutFile,"ChanID %3d : %s : %s\n",
        psuChanInfo->iChanID, psuChanInfo->szChanType, psuChanInfo->szChanName);

    if (psuChanInfo->ulTMATS != 0)
        fprintf(psuOutFile,"    TMATS             %10lu\n",   psuChanInfo->ulTMATS);

    if (psuChanInfo->ulEvents != 0)
        fprintf(psuOutFile,"    Events            %10lu\n",   psuChanInfo->ulEvents);

    if (psuChanInfo->ulIndex != 0)
        fprintf(psuOutFile,"    Index             %10lu\n",   psuChanInfo->ulIndex);

    if (psuChanInfo->ulIrigTime != 0)
        fprintf(psuOutFile,"    IRIG Time         %10lu\n",   psuChanInfo-
>ulIrigTime);

    if ((psuChanInfo->psu1553Info != NULL)  &&
        (psuChanInfo->psu1553Info->ulTotalBusMsgs != 0))
        {

        // Loop through all RT, TR, and SA combinations
        for (lMsgIdx=0; lMsgIdx<0x4000; lMsgIdx++)
            {
            if (psuChanInfo->psu1553Info->aulMsgs[lMsgIdx] != 0)
                {
                fprintf(psuOutFile,"    RT %2d  %c  SA %2d  Msgs %9lu  Errs %9lu\n",
                    (lMsgIdx >>  6) & 0x001f,
                    (lMsgIdx >>  5) & 0x0001 ? 'T' : 'R',
                    (lMsgIdx      ) & 0x001f,
                    psuChanInfo->psu1553Info->aulMsgs[lMsgIdx],
                    psuChanInfo->psu1553Info->aulErrs[lMsgIdx]);
                } // end if count not zero
            } // end for each combination

//      fprintf(psuOutFile,"  Manchester Errors :   %10lu\n", psuChanInfo-
>ulErrManchester);
//      fprintf(psuOutFile,"  Parity Errors     :   %10lu\n", psuChanInfo-
>ulErrParity);
//      fprintf(psuOutFile,"  Overrun Errors    :   %10lu\n", psuChanInfo-
>ulErrOverrun);
//      fprintf(psuOutFile,"  Timeout Errors    :   %10lu\n", psuChanInfo-
>ulErrTimeout);

        if (psuChanInfo->psu1553Info->bRT2RTFound == bTRUE)
            {
            fprintf(psuOutFile,"\n  Warning - RT to RT transfers found in the
data\n");
            if (m_bLogRT2RT == bTRUE)
                fprintf(psuOutFile,"    Message total is NOT the sum of individual RT
totals\n");
            else
                fprintf(psuOutFile,"    Some transmit RTs may not be shown\n");
            } // end if RT to RT

        fprintf(psuOutFile,"    Totals - %ld Message in %ld good IRIG packets, %ld bad
packets\n",
```

```
            psuChanInfo->psu1553Info->ulTotalBusMsgs,
            psuChanInfo->psu1553Info->ulTotalIrigPackets,
            psuChanInfo->psu1553Info->ulTotalIrigPacketErrors);
        } // end if 1553 messages

    if (psuChanInfo->ulPCM != 0)
        fprintf(psuOutFile,"    PCM               %10lu\n",   psuChanInfo->ulPCM);

    if (psuChanInfo->ulAnalog != 0)
        fprintf(psuOutFile,"    Analog            %10lu\n",   psuChanInfo->ulAnalog);

    if (psuChanInfo->paARINC429 != NULL)
        {
        unsigned int    uBus;
        unsigned int    uLabel;
        for (uBus=0; uBus<0x100; uBus++)
            for (uLabel=0; uLabel<0x100; uLabel++)
                {
                if (psuChanInfo->paARINC429->aulMsgs[uBus][uLabel] != 0)
                    fprintf(psuOutFile,"    ARINC 429  Subchan %3u  Label %3o    Msgs
%10lu\n", uBus, uLabel,
                            psuChanInfo->paARINC429->aulMsgs[uBus][uLabel]);
                }
        }


    if (psuChanInfo->ulMPEG2 != 0)
        fprintf(psuOutFile,"    Video             %10lu\n",   psuChanInfo->ulMPEG2);

    if (psuChanInfo->ulUART != 0)
        fprintf(psuOutFile,"    UART              %10lu\n",   psuChanInfo->ulUART);

    if (psuChanInfo->ulEthernet != 0)
        fprintf(psuOutFile,"    Ethernet          %10lu\n",   psuChanInfo-
>ulEthernet);

    if (psuChanInfo->ul16PP194 != 0)
        fprintf(psuOutFile,"    16PP194           %10lu\n",   psuChanInfo->ul16PP194);

    if (psuChanInfo->ulDiscrete != 0)
        fprintf(psuOutFile,"    Discrete          %10lu\n",   psuChanInfo-
>ulDiscrete);

    if (psuChanInfo->ulParallel != 0)
        fprintf(psuOutFile,"    Parallel          %10lu\n",   psuChanInfo-
>ulParallel);

    if (psuChanInfo->ulMessage != 0)
        fprintf(psuOutFile,"    Message Data      %10lu\n",   psuChanInfo->ulMessage);

    if (psuChanInfo->ulImage != 0)
        fprintf(psuOutFile,"    Image             %10lu\n",   psuChanInfo->ulImage);

    if (psuChanInfo->ulTSPI != 0)
        fprintf(psuOutFile,"    TSPI              %10lu\n",   psuChanInfo->ulTSPI);

    if (psuChanInfo->ulCAN != 0)
        fprintf(psuOutFile,"    CAN Bus           %10lu\n",   psuChanInfo->ulCAN);

    if (psuChanInfo->ulFibreChan != 0)
        fprintf(psuOutFile,"    Fibre Channel     %10lu\n",   psuChanInfo-
>ulFibreChan);
```

```
    if (psuChanInfo->ulARINC664 != 0)
        fprintf(psuOutFile,"    ARINC 664         %10lu\n",   psuChanInfo-
>ulARINC664);

    if (psuChanInfo->ulUserDefined != 0)
        fprintf(psuOutFile,"    User Defined      %10lu\n",   psuChanInfo-
>ulUserDefined);

    if (psuChanInfo->ulOther != 0)
        fprintf(psuOutFile,"    Other messages    %10lu\n",   psuChanInfo->ulOther);

    fprintf(psuOutFile,"\n",   psuChanInfo->ulOther);
    return;
    }



/* ---------------------------------------------------------------------- */

void vPrintTmats(SuTmatsInfo * psuTmatsInfo, FILE * psuOutFile)
    {
    int                     iGIndex;
    int                     iRIndex;
//    int                     iRDsiIndex;
    SuGDataSource         * psuGDataSource;
    SuRRecord             * psuRRecord;
    SuRDataSource         * psuRDataSource;

    // Print out the TMATS info
    // ------------------------

    fprintf(psuOutFile,"\n=-=-= Channel Summary =-=-=\n\n");

    // G record
    fprintf(psuOutFile,"Program Name - %s\n",psuTmatsInfo->psuFirstGRecord-
>szProgramName);
    fprintf(psuOutFile,"IRIG 106 Rev - %s\n",psuTmatsInfo->psuFirstGRecord-
>szIrig106Rev);
    fprintf(psuOutFile,"Channel  Type          Data Source         \n");
    fprintf(psuOutFile,"-------  ------------  --------------------\n");

    // Data sources
    psuGDataSource = psuTmatsInfo->psuFirstGRecord->psuFirstGDataSource;
    do  {
        if (psuGDataSource == NULL) break;

        // G record data source info
        iGIndex = psuGDataSource->iIndex;

        // R record info
        psuRRecord = psuGDataSource->psuRRecord;
        do  {
            if (psuRRecord == NULL) break;
            iRIndex = psuRRecord->iIndex;

            // R record data sources
            psuRDataSource = psuRRecord->psuFirstDataSource;
            do  {
                if (psuRDataSource == NULL) break;
//                iRDsiIndex = psuRDataSource->iIndex;
                fprintf(psuOutFile," %5s ",   psuRDataSource->szTrackNumber);
                fprintf(psuOutFile,"  %-12s", psuRDataSource->szChannelDataType);
```

```
                fprintf(psuOutFile,"  %-20s", psuRDataSource->szDataSourceID);
                fprintf(psuOutFile,"\n");
                psuRDataSource = psuRDataSource->psuNext;
                } while (bTRUE);

            psuRRecord = psuRRecord->psuNext;
            } while (bTRUE);


        psuGDataSource = psuTmatsInfo->psuFirstGRecord->psuFirstGDataSource->psuNext;
        } while (bTRUE);

    return;
    }



/* ---------------------------------------------------------------------- */

void vProcessTmats(SuTmatsInfo * psuTmatsInfo, SuChanInfo * apsuChanInfo[])
    {
//    unsigned            uArrayIdx;
//    unsigned char       u1553ChanIdx;
    SuRRecord         * psuRRecord;
    SuRDataSource     * psuRDataSrc;
    int                 iTrackNumber;

    // Find channels mentioned in TMATS record
    psuRRecord   = psuTmatsInfo->psuFirstRRecord;
    while (psuRRecord != NULL)
        {
        // Get the first data source for this R record
        psuRDataSrc = psuRRecord->psuFirstDataSource;
        while (psuRDataSrc != NULL)
            {
            iTrackNumber = atoi(psuRDataSrc->szTrackNumber);

            // Make sure a message count structure exists
            if (apsuChanInfo[iTrackNumber] == NULL)
                {
                apsuChanInfo[iTrackNumber] = malloc(sizeof(SuChanInfo));
                memset(apsuChanInfo[iTrackNumber], 0, sizeof(SuChanInfo));
                apsuChanInfo[iTrackNumber]->iChanID = iTrackNumber;
                }

            // Now save channel type and name
            strcpy(apsuChanInfo[iTrackNumber]->szChanType, psuRDataSrc-
>szChannelDataType);
            strcpy(apsuChanInfo[iTrackNumber]->szChanName, psuRDataSrc-
>szDataSourceID);

            // Get the next R record data source
            psuRDataSrc = psuRDataSrc->psuNext;
            } // end while walking R data source linked list

        // Get the next R record
        psuRRecord = psuRRecord->psuNext;

        } // end while walking R record linked list

    return;
    }
```

```
/* ---------------------------------------------------------------------- */

// The ARINC 429 label field is bit reversed. This array maps the ARINC
// label field to its non-reversed bretheren.

void vMakeArincLabelMap(unsigned char m_aArincLabelMap[])
    {
    unsigned int    uLabelIdx;
    unsigned char   uRLabel;
    unsigned char   uLabel;
    int             iBitIdx;

    for (uLabelIdx=0; uLabelIdx<0x100; uLabelIdx++)
        {
        uLabel = (unsigned char)uLabelIdx;
        uRLabel = 0;
        for (iBitIdx=0; iBitIdx<8; iBitIdx++)
            {
            uRLabel <<= 1;
            uRLabel  |= uLabel & 0x01;
            uLabel  >>= 1;
            } // end for each bit in the label
        m_aArincLabelMap[uLabelIdx] = uRLabel;
        } // end for each label

    }



/* ---------------------------------------------------------------------- */

void vUsage(void)
    {
    printf("\nI106STAT "MAJOR_VERSION"."MINOR_VERSION" "__DATE__" "__TIME__"\n");
    printf("Print totals by channel and message type from a Ch 10 data file\n");
    printf("Freeware Copyright (C) 2006 Irig106.org\n\n");
    printf("Usage: i106stat <input file> <output file> [flags]\n");
    printf("   <filename> Input/output file names\n");
    printf("   -r         Log both sides of RT to RT transfers\n");
    printf("   -v         Verbose\n");
    }
```

Appendix D

Example Program – Decode TMATS

The following software program opens a Chapter 10 file for reading, extracts the TMATS setup record, and displays it one of several ways. It demonstrates reading a Chapter 9 TMATS packets, and parsing the TMATS attributes.

```
/*===========================================================================

  idmptmat - Read and dump a TMATS record from an IRIG 106 Ch 10 data file

 Copyright (c) 2006 Irig106.org

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

***************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>

#include "stdint.h"

#include "irig106ch10.h"
#include "i106_decode_tmats.h"


/*
 * Macros and definitions
 * ----------------------
```

```
 */

#define MAJOR_VERSION  "01"
#define MINOR_VERSION  "01"

#if !defined(bTRUE)
#define bTRUE   (1==1)
#define bFALSE  (1==0)
#endif


/*
 * Module data
 * -----------
 */


/*
 * Function prototypes
 * -------------------
 */

void    vDumpRaw(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile);
void    vDumpTree(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile);
void    vDumpChannel(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile);
void    vUsage(void);


/* ------------------------------------------------------------------------ */

int main(int argc, char ** argv)
    {

    char                    szInFile[80];     // Input file name
    char                    szOutFile[80];    // Output file name
    int                     iArgIdx;
    FILE                  * ptOutFile;        // Output file handle
    int                     bRawOutput;
    int                     bTreeOutput;
    int                     bChannelOutput;
    unsigned long           ulBuffSize = 0L;

    int                     iI106Ch10Handle;
    EnI106Status            enStatus;
    SuI106Ch10Header        suI106Hdr;

    unsigned char         * pvBuff = NULL;


/* Make sure things stay on UTC */

    putenv("TZ=GMT0");
    tzset();

/*
 * Process the command line arguements
 */

    if (argc < 2) {
    vUsage();
    return 1;
    }
```

```
    bRawOutput     = bFALSE;    // No verbosity
    bTreeOutput    = bFALSE;
    bChannelOutput = bFALSE;
    szInFile[0]    = '\0';
    strcpy(szOutFile,"");       // Default is stdout

    for (iArgIdx=1; iArgIdx<argc; iArgIdx++)
        {

        switch (argv[iArgIdx][0])
            {

            // Handle command line flags
            case '-' :
                switch (argv[iArgIdx][1])
                    {

                    case 'r' :                    // Raw output
                        bRawOutput = bTRUE;
                        break;

                    case 't' :                    // Tree output
                        bTreeOutput = bTRUE;
                        break;

                    case 'c' :                    // Channel summary
                        bChannelOutput = bTRUE;
                        break;

                    default :
                        break;
                    } // end flag switch
                break;

            // Anything else must be a file name
            default :
                if (szInFile[0] == '\0') strcpy(szInFile, argv[iArgIdx]);
                else                     strcpy(szOutFile,argv[iArgIdx]);
                break;

            } // end command line arg switch
        } // end for all arguments

    if (strlen(szInFile)==0)
        {
        vUsage();
        return 1;
        }

    // Make sure at least on output is turned on
    if ((bRawOutput     == bFALSE) &&
        (bTreeOutput    == bFALSE) &&
        (bChannelOutput == bFALSE))
        bChannelOutput = bTRUE;

/*
 * Opening banner
 * --------------
 */

    fprintf(stderr, "\nIDMPTMAT "MAJOR_VERSION"."MINOR_VERSION"\n");
    fprintf(stderr, "Freeware Copyright (C) 2006 Irig106.org\n\n");
```

```
/*
 * Open file and get everything init'ed
 * ------------------------------------
 */

    // Open file and allocate a buffer for reading data.
    enStatus = enI106Ch10Open(&iI106Ch10Handle, szInFile, I106_READ);
    switch (enStatus)
        {
        case I106_OPEN_WARNING :
            fprintf(stderr, "Warning opening data file : Status = %d\n", enStatus);
            break;
        case I106_OK :
            break;
        default :
            fprintf(stderr, "Error opening data file : Status = %d\n", enStatus);
            return 1;
            break;
        }

    // If output file specified then open it
    if (strlen(szOutFile) != 0)
        {
        ptOutFile = fopen(szOutFile,"w");
        if (ptOutFile == NULL)
            {
            fprintf(stderr, "Error opening output file\n");
            return 1;
            }
        }

    // No output file name so use stdout
    else
        {
        ptOutFile = stdout;
        }

/*
 * Read the TMATS record
 */

    // Read the next header
    enStatus = enI106Ch10ReadNextHeader(iI106Ch10Handle, &suI106Hdr);

    if (enStatus != I106_OK)
        {
        fprintf(stderr, " Error reading header : Status = %d\n", enStatus);
        return 1;
        }

    // Make sure our buffer is big enough, size *does* matter
    if (ulBuffSize < uGetDataLen(&suI106Hdr))
        {
        pvBuff = realloc(pvBuff, uGetDataLen(&suI106Hdr));
        ulBuffSize = uGetDataLen(&suI106Hdr);
        }

    // Read the data buffer
    enStatus = enI106Ch10ReadData(iI106Ch10Handle, ulBuffSize, pvBuff);
    if (enStatus != I106_OK)
        {
        fprintf(stderr, " Error reading data : Status = %d\n", enStatus);
        return 1;
```

```
        }

    if (suI106Hdr.ubyDataType != I106CH10_DTYPE_TMATS)
        {
        fprintf(stderr, " Error reading data : first message not TMATS");
        return 1;
        }

    // Generate output
    fprintf(ptOutFile, "IDMPTMAT "MAJOR_VERSION"."MINOR_VERSION"\n");
    fprintf(ptOutFile, "TMATS from file %s\n\n", szInFile);

    if (bRawOutput == bTRUE)
        vDumpRaw(&suI106Hdr, pvBuff, ptOutFile);

    if (bTreeOutput == bTRUE)
        vDumpTree(&suI106Hdr, pvBuff, ptOutFile);

    if (bChannelOutput == bTRUE)
        vDumpChannel(&suI106Hdr, pvBuff, ptOutFile);

    // Done so clean up
    free(pvBuff);
    pvBuff = NULL;

    fclose(ptOutFile);

    return 0;
    }


/* ---------------------------------------------------------------------- */

// Output the raw, unformated TMATS record

void  vDumpRaw(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile)
    {
    unsigned long    lChrIdx;
    char           * achBuff = pvBuff;

    for (lChrIdx = 0; lChrIdx<psuI106Hdr->ulDataLen; lChrIdx++)
        fputc(achBuff[lChrIdx], ptOutFile);

    return;
    }



/* ---------------------------------------------------------------------- */

void vDumpTree(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile)
    {
    EnI106Status            enStatus;
    int                     iGIndex;
    int                     iRIndex;
    int                     iRDsiIndex;
    SuTmatsInfo             suTmatsInfo;
    SuGDataSource         * psuGDataSource;
    SuRRecord             * psuRRecord;
    SuRDataSource         * psuRDataSource;

    // Process the TMATS info
    enStatus = enI106_Decode_Tmats(psuI106Hdr, pvBuff, &suTmatsInfo);
```

```
if (enStatus != I106_OK)
    {
    fprintf(stderr, " Error processing TMATS record : Status = %d\n", enStatus);
    return;
    }

// Print out the TMATS info
// ------------------------

// G record
fprintf(ptOutFile,
    "(G) Program Name - %s\n",suTmatsInfo.psuFirstGRecord->szProgramName);
fprintf(ptOutFile,
    "(G) IRIG 106 Rev - %s\n",suTmatsInfo.psuFirstGRecord->szIrig106Rev);

// Data sources
psuGDataSource = suTmatsInfo.psuFirstGRecord->psuFirstGDataSource;
do  {
    if (psuGDataSource == NULL) break;

    // G record data source info
    iGIndex = psuGDataSource->iDataSourceNum;
    fprintf(ptOutFile, "  (G\\DSI-%i) Data Source ID   - %s\n",
        psuGDataSource->iDataSourceNum,
        suTmatsInfo.psuFirstGRecord->psuFirstGDataSource->szDataSourceID);
    fprintf(ptOutFile, "  (G\\DST-%i) Data Source Type - %s\n",
        psuGDataSource->iDataSourceNum,
        suTmatsInfo.psuFirstGRecord->psuFirstGDataSource->szDataSourceType);

    // R record info
    psuRRecord = psuGDataSource->psuRRecord;
    do  {
        if (psuRRecord == NULL) break;
        iRIndex = psuRRecord->iRecordNum;
        fprintf(ptOutFile, "    (R-%i\\ID) Data Source ID - %s\n",
            iRIndex, psuRRecord->szDataSourceID);

        // R record data sources
        psuRDataSource = psuRRecord->psuFirstDataSource;
        do  {
            if (psuRDataSource == NULL) break;
            iRDsiIndex = psuRDataSource->iDataSourceNum;
            fprintf(ptOutFile,
                    "      (R-%i\\DSI-%i) Data Source ID - %s\n",
                    iRIndex, iRDsiIndex, psuRDataSource->szDataSourceID);
            fprintf(ptOutFile,
                    "      (R-%i\\DST-%i) Channel Type   - %s\n",
                    iRIndex, iRDsiIndex, psuRDataSource->szChannelDataType);
            fprintf(ptOutFile,
                    "      (R-%i\\TK1-%i) Track Number   - %i\n",
                    iRIndex, iRDsiIndex, psuRDataSource->iTrackNumber);
            psuRDataSource = psuRDataSource->psuNextRDataSource;
            } while (bTRUE);

        psuRRecord = psuRRecord->psuNextRRecord;
        } while (bTRUE);


    psuGDataSource =
        suTmatsInfo.psuFirstGRecord->psuFirstGDataSource->psuNextGDataSource;
    } while (bTRUE);
```

```
    return;
    }


/* ---------------------------------------------------------------------- */

void vDumpChannel(SuI106Ch10Header * psuI106Hdr, void * pvBuff, FILE * ptOutFile)
    {
    EnI106Status            enStatus;
    int                     iGIndex;
    int                     iRIndex;
    int                     iRDsiIndex;
    SuTmatsInfo             suTmatsInfo;
    SuGDataSource         * psuGDataSource;
    SuRRecord             * psuRRecord;
    SuRDataSource         * psuRDataSource;

    // Process the TMATS info
    enStatus = enI106_Decode_Tmats(psuI106Hdr, pvBuff, &suTmatsInfo);
    if (enStatus != I106_OK)
        {
        fprintf(stderr, " Error processing TMATS record : Status = %d\n", enStatus);
        return;
        }

    // Print out the TMATS info
    // ------------------------

    // G record
    fprintf(ptOutFile,
        "Program Name - %s\n",suTmatsInfo.psuFirstGRecord->szProgramName);
    fprintf(ptOutFile,
        "IRIG 106 Rev - %s\n",suTmatsInfo.psuFirstGRecord->szIrig106Rev);
    fprintf(ptOutFile,
        "Channel  Type          Enabled   Data Source         \n");
    fprintf(ptOutFile,
        "-------  ------------  --------  --------------------\n");

    // Data sources
    psuGDataSource = suTmatsInfo.psuFirstGRecord->psuFirstGDataSource;
    do  {
        if (psuGDataSource == NULL) break;

        // G record data source info
        iGIndex = psuGDataSource->iDataSourceNum;

        // R record info
        psuRRecord = psuGDataSource->psuRRecord;
        do  {
            if (psuRRecord == NULL) break;
            iRIndex = psuRRecord->iRecordNum;

            // R record data sources
            psuRDataSource = psuRRecord->psuFirstDataSource;
            do  {
                if (psuRDataSource == NULL) break;
                iRDsiIndex = psuRDataSource->iDataSourceNum;
                fprintf(ptOutFile, " %5i ",   psuRDataSource->iTrackNumber);
                fprintf(ptOutFile, "  %-12s", psuRDataSource->szChannelDataType);
                fprintf(ptOutFile, "  %-8s",  psuRDataSource->bEnabled ? "En" :
"Dis");
                fprintf(ptOutFile, "  %-20s", psuRDataSource->szDataSourceID);
```

```
                fprintf(ptOutFile, "\n");
                psuRDataSource = psuRDataSource->psuNextRDataSource;
                } while (bTRUE);

            psuRRecord = psuRRecord->psuNextRRecord;
            } while (bTRUE);


        psuGDataSource =
            suTmatsInfo.psuFirstGRecord->psuFirstGDataSource->psuNextGDataSource;
        } while (bTRUE);

    return;
    }



/* ---------------------------------------------------------------------- */

void vUsage(void)
    {
    printf("\nIDMPTMAT - IDMPTMAT "MAJOR_VERSION"."MINOR_VERSION" "__DATE__"
"__TIME__"\n");
    printf("Read and output TMATS record from a Ch 10 data file\n");
    printf("Freeware Copyright (C) 2006 Irig106.org\n\n");
    printf("Usage: idmptmat <infile> <outfile> <flags>\n");
    printf("  -c      Output channel summary format (default)\n");
    printf("  -t      Output tree view format\n");
    printf("  -r      Output raw TMATS\n");
    return;
    }
```

Appendix E

Example C Program – Display Channel Details

The following software program opens a Chapter 10 file for reading and displays channel ID and data type for each channel as well as a count of how many packets are present for each channel in a chapter 10 file.

```
/*============================================================================

  stat.c - Display some basic information about the channels within a Chapter 10
    file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#include "irig106ch10.h"
#include "stat_args.c"
#include "common.h"

typedef struct {
    unsigned int type;
    int id;
    int packets;
```

```
} ChanSpec;


int main(int argc, char ** argv){

    // Get commandline args.
    DocoptArgs args = docopt(argc, argv, 1, "1");
    if (args.help){
        printf(args.help_message);
        return 1;
    }
    else if (argc < 2){
        printf(args.usage_pattern);
        return 1;
    }

    SuI106Ch10Header header;
    int packets = 0, input_handle;
    float byte_size = 0.0;
    static ChanSpec * channels[0x10000];

    // Open the source file.
    EnI106Status status = enI106Ch10Open(&input_handle, argv[1], I106_READ);
    if (status != I106_OK){
        char msg[200] = "Error opening file ";
        strcat(msg, argv[1]);
        return error(msg);
    }

    // Iterate over selected packets (based on args).
    while (1){
        status = enI106Ch10ReadNextHeader(input_handle, &header);

        // Exit once file ends.
        if (status != I106_OK){
            break;
        }

        // Ensure that we're interested in this particular packet.
        if (args.exclude && match(header.uChID, args.exclude)){
            continue;
        }
        else if (args.channel && !match(header.uChID, args.channel)){
            continue;
        }
        else if (args.type && !match(header.ubyDataType, args.type)){
            continue;
        }

        // Increment overall size and packet counts.
        byte_size += header.ulPacketLen;
        packets++;

        // Find the channel info based on ID and type.
        int i = 0;
        for (i; i < 0x10000; i++){

            // Create a listing if none exists.
            if (channels[i] == NULL){
                channels[i] = (ChanSpec *)malloc(sizeof(ChanSpec));
                memset(channels[i], 0, sizeof(ChanSpec));
                channels[i]->id = (unsigned int)header.uChID;
                channels[i]->type = (unsigned int)header.ubyDataType;
```

```
                channels[i]->packets = 0;
            }

            if (channels[i]->id == (unsigned int)header.uChID && channels[i]->type ==
(unsigned int)header.ubyDataType){
                break;
            }
        }

        // Increment the counter.
        channels[i]->packets++;

    }

    // Print details for each channel.
    printf("Channel ID      Data Type%35sPackets\n", "");
    printf("-------------------------------------------------------------------------------
------\n");
    int i = 0;
    while (channels[i] != NULL){
        printf("Channel%3d", channels[i]->id);
        printf("%6s", "");
        printf("0x%-34x", channels[i]->type);
        printf("%7d packets", channels[i]->packets);
        printf("\n");
        i++;
    }

    // Find a more readable size unit than bytes.
    char *unit = "b";
    if (byte_size > 1024){
        byte_size /= 1024;
        unit = "kb";
    }
    if (byte_size > 1024){
        byte_size /= 1024;
        unit = "mb";
    }
    if (byte_size > 1024){
        byte_size /= 1024;
        unit = "gb";
    }

    // Print file summary.
    printf("-------------------------------------------------------------------------------
------\n");
    printf("Summary for %s:\n", argv[1]);
    printf("    Size: %.*f%s\n", 2, byte_size, unit);
    printf("    Packets: %d\n", packets);
    printf("    Channels: %d\n", i);

    return quit(0);
}
```

This page intentionally left blank.

Appendix F

Example Python Program – Display Channel Details

The following software program opens a Chapter 10 file for reading and displays channel ID and data type for each channel as well as a count of how many packets are present for each channel in a Chapter 10 file. This example uses the Python wrapper to the irig106lib library.

```
#!/usr/bin/env python

"""
  stat.py - Display some basic information about the channels within a
    Chapter 10 file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: stat.py <file> [options]

Options:
    -c CHANNEL..., --channel CHANNEL... Specify channels to include(csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to show (csv, may \
be decimal or hex eg: 0x40)."""

from contextlib import closing

from docopt import docopt
from Py106.Packet import IO, FileMode, Status, DataType

from walk import walk_packets
```

```
if __name__ == '__main__':

    # Get commandline args.
    args = docopt(__doc__)

    channels, packets, size = ([], 0, 0)

    # Open the source file.
    with closing(IO()) as PktIO:
        RetStatus = PktIO.open(args['<file>'], FileMode.READ)
        if RetStatus != Status.OK:
            print "Error opening data file %s" % args['<file>']
            raise SystemExit

        # Iterate over selected packets (based on args).
        for packet in walk_packets(PktIO.packet_headers(), args):

            # Increment overall size and packet count.
            size += packet.PacketLen
            packets += 1

            # Find the channel info based on ID and type.
            channel_index = None
            for i, channel in enumerate(channels):
                if channel['id'] == packet.ChID and \
                        channel['type'] == packet.DataType:
                    channel_index = i
                    break

            # Find the channel info based on ID and type.
            if channel_index is None:
                channel_index = len(channels)
                channels.append({'packets': 0,
                                 'type': packet.DataType,
                                 'id': packet.ChID})

            # Increment the counter.
            channels[channel_index]['packets'] += 1

    # Print details for each channel.
    print('Channel ID     Data Type' + 'Packets'.rjust(46))
    print('-' * 80)
    for channel in channels:
        dtype = DataType.name(channel['type'])
        print (''.join((('Channel %s' % channel['id']).ljust(15),
                        ('%s - %s' % (hex(channel['type']), dtype)).ljust(35),
                        ('%s packets' % channel['packets']).rjust(20))))

    # Find a more readable size unit than bytes.
    units = ['gb', 'mb', 'kb']
    unit = 'b'
    while size > 1024 and units:
        size /= 1024.0
        unit = units.pop()

    # Print file summary.
    print('-' * 80)
    print('Summary for %s:' % args['<file>'])
    print('    Size: %s %s' % (round(size, 2), unit))
    print('    Packets: %s' % packets)
    print('    Channels: %s' % len(channels))
```

Appendix G

Example Python Program – Display Channel Details

The following software program opens a Chapter 10 file for reading and displays channel ID and data type for each channel as well as a count of how many packets are present for each channel in a chapter 10 file. This example uses the PyChapter10 library.

```
#!/usr/bin/env python

"""
  stat.py - Display some basic information about the channels within a
    Chapter 10 file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: stat.py <file> [options]

Options:
    -c CHANNEL..., --channel CHANNEL... Specify channels to include(csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to show (csv, may \
be decimal or hex eg: 0x40)."""

from docopt import docopt

from chapter10 import C10
from chapter10.datatypes import get_label

from walk import walk_packets
```

```
if __name__ == '__main__':

    # Get commandline args.
    args = docopt(__doc__)

    channels, packets, size = ([], 0, 0)

    # Open the source file.
    src = C10(args['<file>'])

    # Iterate over selected packets (based on args).
    for packet in walk_packets(src, args):

        # Increment overall size and packet count.
        size += packet.packet_length
        packets += 1

        # Find the channel info based on ID and type.
        channel_index = None
        for i, channel in enumerate(channels):
            if channel['id'] == packet.channel_id and \
                    channel['type'] == packet.data_type:
                channel_index = i
                break

        # Create a listing if none exists.
        if channel_index is None:
            channel_index = len(channels)
            channels.append({'packets': 0,
                             'type': packet.data_type,
                             'id': packet.channel_id})

        # Increment the counter.
        channels[channel_index]['packets'] += 1

    # Print details for each channel.
    print('Channel ID     Data Type' + 'Packets'.rjust(46))
    print('-' * 80)
    for channel in channels:
        print (''.join((('Channel %s' % channel['id']).ljust(15),
                        ('%s - %s' % (hex(channel['type']),
                                      get_label(channel['type']))).ljust(35),
                        ('%s packets' % channel['packets']).rjust(20))))

    # Find a more readable size unit than bytes.
    units = ['gb', 'mb', 'kb']
    unit = 'b'
    while size > 1024 and units:
        size /= 1024.0
        unit = units.pop()

    # Print file summary.
    print('-' * 80)
    print('Summary for %s:' % args['<file>'])
    print('    Size: %s %s' % (round(size, 2), unit))
    print('    Packets: %s' % packets)
    print('    Channels: %s' % len(channels))
```

Appendix H

Example C Program – Copy Chapter 10 File

The following software program opens a Chapter 10 file for reading and copies its data to another file optionally filtering by channel ID and/or data type.

```
/*============================================================================

  copy.c - Copy all or part of a Chapter 10 file based on data types, channels,
    etc.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#include "irig106ch10.h"
#include "copy_args.c"
#include "common.h"

int main(int argc, char ** argv){

    // Get commandline args.
    DocoptArgs args = docopt(argc, argv, 1, "1");
    if (argc < 3){
        printf(args.usage_pattern);
        return quit(1);
```

```
}

// Open input and output files.
int input_handle;
EnI106Status status = enI106Ch10Open(&input_handle, argv[1], I106_READ);
if (status != I106_OK){
    char msg[200] = "Error opening source file: ";
    strcat(msg, argv[1]);
    return error(msg);
}
FILE * output = fopen(argv[2], "wb");
if (output == NULL){
    return error("Couldn't open destination file.");
}

SuI106Ch10Header header;
void * buffer = malloc(24);

// Copy TMATS
status = enI106Ch10ReadNextHeader(input_handle, &header);
if (status != I106_OK){
    printf("Finished");
    return quit(0);
}
buffer = realloc(buffer, header.ulPacketLen);
status = enI106Ch10ReadDataFile(input_handle, header.ulPacketLen, buffer);
if (status != I106_OK){
    printf("Error reading TMATS.");
    return quit(0);
}
int header_len = 24;
if (header.ubyPacketFlags & (0x1 << 7)){
    header_len = 36;
}
fwrite(&header, header_len, 1, output);
fwrite(buffer, header.ulPacketLen - header_len, 1, output);

// Iterate over packets based on args.
while (1){

    // Read next header or exit.
    status = enI106Ch10ReadNextHeader(input_handle, &header);
    if (status != I106_OK){
        printf("Finished");
        return quit(0);
    }

    // Ensure that we're interested in this particular packet.
    if (args.exclude && match(header.uChID, args.exclude)){
        continue;
    }
    else if (args.channel && !match(header.uChID, args.channel)){
        continue;
    }
    else if (args.type && !match(header.ubyDataType, args.type)){
        continue;
    }

    // Copy packet to new file.
    buffer = realloc(buffer, header.ulPacketLen);
    status = enI106Ch10ReadDataFile(input_handle, header.ulPacketLen, buffer);
    if (status != I106_OK){
        printf("Error reading packet.");
```

```
            continue;
        }
        header_len = 24;
        if (header.ubyPacketFlags & (0x1 << 7)){
            header_len = 36;
        }
        fwrite(&header, header_len, 1, output);
        fwrite(buffer, header.ulPacketLen, 1, output);
    }

}
```

This page intentionally left blank.

Appendix I

Example Python Program – Copy Chapter 10 File

The following software program opens a Chapter 10 file for reading and copies its data to another file optionally filtering by channel ID and/or data type. This sample uses the Python wrapper to the irig106lib C library.

```
#!/usr/bin/env python

"""
  copy.py - Copy all or part of a Chapter 10 file based on data types,
    channels, etc.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: copy.py <src> <dst> [options]

Options:
    -c CHANNEL..., --channel CHANNEL... Specify channels to include (csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to copy (csv, may\
 be decimal or hex eg: 0x40)
    -f --force                          Overwrite existing files."""

from contextlib import closing
import os

from docopt import docopt

from Py106.Packet import IO, FileMode, Status
```

```
from walk import walk_args


if __name__ == '__main__':

    # Get commandline args.
    args = docopt(__doc__)

    # Don't overwrite unless explicitly required.
    if os.path.exists(args['<dst>']) and not args['--force']:
        print('dst file already exists. Use -f to overwrite.')
        raise SystemExit

    # Open input and output files.
    with open(args['<dst>'], 'wb') as out, closing(IO()) as PktIO:
        RetStatus = PktIO.open(args['<src>'], FileMode.READ)
        if RetStatus != Status.OK:
            print "Error opening data file %s" % args['<src>']
            raise SystemExit

        # Iterate over packets based on args.
        channels, exclude, types = walk_args(args)
        i = 0
        while RetStatus == Status.OK:
            RetStatus = PktIO.read_next_header()
            if i > 1:
                if PktIO.read_data() != Status.OK:
                    continue
                elif channels and str(PktIO.Header.ChID) not in channels:
                    continue
                elif str(PktIO.Header.ChID) in exclude:
                    continue
                elif types and PktIO.Header.DataType not in types:
                    continue

            # Copy packet to new file.
            header = buffer(PktIO.Header)[:]
            if not bool(PktIO.Header.PacketFlags & (1 << 7)):
                header = header[:-12]
            out.write(header)
            out.write(PktIO.Buffer.raw[:PktIO.Header.PacketLen - len(header)])

            i += 1
```

Appendix J

Example Python Program – Copy Chapter 10 File

The following software program opens a Chapter 10 file for reading and copies its data to another file optionally filtering by channel ID and/or data type. This sample uses the PyChapter10 library.

```
#!/usr/bin/env python

"""
  copy.py - Copy all or part of a Chapter 10 file based on data types,
    channels, etc.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: copy.py <src> <dst> [options]

Options:
    -c CHANNEL..., --channel CHANNEL... Specify channels to include (csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to copy (csv, may\
 be decimal or hex eg: 0x40)
    -f --force                          Overwrite existing files."""

import os

from chapter10 import C10
from docopt import docopt

from walk import walk_packets
```

```
if __name__ == '__main__':

    # Get commandline args.
    args = docopt(__doc__)

    # Don't overwrite unless explicitly required.
    if os.path.exists(args['<dst>']) and not args['--force']:
        print('dst file already exists. Use -f to overwrite.')
        raise SystemExit

    # Open input and output files.
    with open(args['<dst>'], 'wb') as out:
        src = C10(args['<src>'])

        # Iterate over packets based on args.
        for packet in walk_packets(src, args):

            # Copy packet to new file.
            raw = bytes(packet)
            if len(raw) == packet.packet_length:
                out.write(raw)
```

Appendix K

Example C Program – Export Chapter 10 Data

The following software program opens a Chapter 10 file for reading and writes channel data to separate files. The channels may be filtered by ID or data type.

```
/*============================================================================

  dump.c - Export channel data based on channel ID or data type.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <intrin.h>

#include "irig106ch10.h"
#include "dump_args.c"
#include "common.h"

// Byteswap a buffer.
void swap(char *p, int len) {
    char tmp;
    for (int i = 0; i < ((len / 2)); i++) {
        tmp = p[i * 2];
        p[i * 2] = p[(i * 2) + 1];
        p[(i * 2) + 1] = tmp;
```

```
    }
}

int main(int argc, char ** argv){
    DocoptArgs args = docopt(argc, argv, 1, "1");
    int input_handle;
    SuI106Ch10Header header;
    int packets = 0;
    char * buffer = malloc(24);
    char * ts = malloc(188);
    FILE *out[0x10000];

    // Initialize out to NULLs.
    for (int i = 0; i < 0x10000; i++){
        out[i] = NULL;
    }

    // Validate arguments and offer help.
    if (argc < 2){
        printf(args.usage_pattern);
        return quit(1);
    }

    // Open file for reading.
    EnI106Status status = enI106Ch10Open(&input_handle, argv[1], I106_READ);
    if (status != I106_OK){
        char msg[200] = "Error opening source file: ";
        strcat(msg, argv[1]);
        return error(msg);
    }

    // Parse loop.
    while (1){

        // Read next header or exit.
        status = enI106Ch10ReadNextHeader(input_handle, &header);
        if (status != I106_OK){
            printf("Finished");
            return quit(0);
        }

        // Ensure that we're interested in this particular packet.
        if (args.exclude && match(header.uChID, args.exclude)){
            continue;
        }
        else if (args.channel && !match(header.uChID, args.channel)){
            continue;
        }
        else if (args.type && !match(header.ubyDataType, args.type)){
            continue;
        }

        // Get the correct filename for this channel.
        char filename[10000];
        strcpy(filename, args.output);
        char channel[5];
        sprintf(channel, "/%d", header.uChID);
        strcat(filename, channel);

        // Check for video (byte-swap required)
        if (0x3F < header.ubyDataType < 0x43){
            strcat(filename, ".mpg");
        }
```

```
        // Ensure an output file is open for this channel.
        if (out[header.uChID] == NULL){
            out[header.uChID] = fopen(filename, "wb");

            if (out[header.uChID] == NULL){
                printf("Error opening output file: %s", filename);
                return quit(1);
            }
        }

        // Read packet data.
        buffer = realloc(buffer, header.ulPacketLen);
        status = enI106Ch10ReadDataFile(input_handle, header.ulPacketLen, buffer);
        if (status != I106_OK){
            printf("Error reading packet.");
            continue;
        }

        // Ignore first 4 bytes (CSDW)
        int datalen = header.ulDataLen - 4;
        if (0x3F < header.ubyDataType < 0x43){
            for (int i = 0; i < (datalen / 188); i++){
                memcpy(ts, buffer + 4 + (i * 188), 188);
                swap(ts, 188);
                fwrite(ts, 1, 188, out[header.uChID]);
            }
        }
        else {
            fwrite(buffer + 4, 1, datalen, out[header.uChID]);
        }

    }

    return quit(0);
}
```

This page intentionally left blank.

Appendix L

Example Python Program – Export Chapter 10 Data

The following software program opens a Chapter 10 file for reading and writes channel data to separate files. The channels may be filtered by ID or data type. This sample uses the Python wrapper to the irig106lib C library.

```
#!/usr/bin/env python

"""
  dump.py - Export channel data based on channel ID or data type.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: dump.py <file> [options]

Options:
    -o OUT, --output OUT                The directory to place files \
[default: .].
    -c CHANNEL..., --channel CHANNEL... Specify channels to include(csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to export (csv, may\
 be decimal or hex eg: 0x40)
    -f, --force                         Overwrite existing files."""

from array import array
from contextlib import closing
import atexit
import os
```

```
from docopt import docopt

from Py106.Packet import IO, FileMode, Status
from walk import walk_args


if __name__ == '__main__':
    args = docopt(__doc__)

    # Ensure OUT exists.
    if not os.path.exists(args['--output']):
        os.makedirs(args['--output'])

    channels, exclude, types = walk_args(args)

    # Open input file.
    with closing(IO()) as PktIO:
        RetStatus = PktIO.open(args['<file>'], FileMode.READ)
        if RetStatus != Status.OK:
            print "Error opening data file %s" % args['<file>']
            raise SystemExit

        out = {}

        # Iterate over packets based on args.
        while RetStatus == Status.OK:
            RetStatus = PktIO.read_next_header()

            if channels and str(PktIO.Header.ChID) not in channels:
                continue
            elif str(PktIO.Header.ChID) in exclude:
                continue
            elif types and PktIO.Header.DataType not in types:
                continue

            if PktIO.read_data() != Status.OK:
                continue

            # Get filename for this channel based on data type.
            filename = os.path.join(args['--output'], str(PktIO.Header.ChID))
            if 0x3F < PktIO.Header.DataType < 0x43:
                filename += '.mpg'

            # Ensure a file is open (and will close) for a given channel.
            if filename not in out:

                # Don't overwrite unless explicitly required.
                if os.path.exists(filename) and not args['--force']:
                    print('%s already exists. Use -f to overwrite.' % filename)
                    break

                out[filename] = open(filename, 'wb')
                atexit.register(out[filename].close)

            data = PktIO.Buffer.raw[4:PktIO.Header.PacketLen - 4]

            # Handle special case for video data.
            if bool(0x3F < PktIO.Header.DataType < 0x43):
                for i in range(len(data) / 188):
                    body = array('H', data[i * 188:(i + 1) * 188])
                    body.byteswap()
                    out[filename].write(body.tostring())
```

```
else:
    # Write out raw packet body.
    out[filename].write(data)
```

This page intentionally left blank.

Appendix M

Example Python Program – Export Chapter 10 Data

The following software program opens a Chapter 10 file for reading and writes channel data to separate files. The channels may be filtered by ID or data type. This sample uses the PyChapter10 library.

```
#!/usr/bin/env python

"""
  dump.py - Export channel data based on channel ID or data type.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: dump.py <file> [options]

Options:
    -o OUT, --output OUT                The directory to place files \
[default: .].
    -c CHANNEL..., --channel CHANNEL... Specify channels to include(csv).
    -e CHANNEL..., --exclude CHANNEL... Specify channels to ignore (csv).
    -t TYPE, --type TYPE                The types of data to export (csv, may\
 be decimal or hex eg: 0x40)
    -f, --force                         Overwrite existing files."""

import atexit
import os

from chapter10 import C10, datatypes
from docopt import docopt
```

```
from walk import walk_packets


if __name__ == '__main__':

    # Get commandline args.
    args = docopt(__doc__)

    # Ensure OUT exists.
    if not os.path.exists(args['--output']):
        os.makedirs(args['--output'])

    out = {}

    # Iterate over packets based on args.
    for packet in walk_packets(C10(args['<file>']), args):

        # Get filename for this channel based on data type.
        filename = os.path.join(args['--output'], str(packet.channel_id))
        t, f = datatypes.format(packet.data_type)
        if t == 0 and f == 1:
            filename += packet.body.frmt == 0 and '.tmats' or '.xml'
        elif t == 8:
            filename += '.mpg'

        # Ensure a file is open (and will close) for a given channel.
        if filename not in out:

            # Don't overwrite unless explicitly required.
            if os.path.exists(filename) and not args['--force']:
                print('%s already exists. Use -f to overwrite.' % filename)
                break

            out[filename] = open(filename, 'wb')
            atexit.register(out[filename].close)

        # Only write TMATS once.
        elif t == 0 and f == 1:
            continue

        # Handle special case for video data.
        if t == 8:
            data = b''.join([p.data for p in packet.body.mpeg])
        else:
            data = packet.body.data

        # Write out raw packet body.
        out[filename].write(data)
```

Appendix N

Example C Program – Reindex Chapter 10 Files

The following software program opens a Chapter 10 file for reading and writes its content to another file while stripping and (optionally) rebuilding index packets.

```
/*=============================================================================

  reindex.c - Strip and (optionally) rebuild index packets for a Chapter 10
    file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.

*============================================================================*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>

#include "irig106ch10.h"
#include "reindex_args.c"
#include "common.h"

void gen_node(int64_t offset, SuI106Ch10Header * packets, int packets_c, int64_t
offsets[], FILE * output, int seq){
    printf("Index node for %d packets\n", packets_c);

    // Packet header
    SuI106Ch10Header index_header;
    index_header.uSync = 0xeb25;
```

```
    index_header.uChID = 0;
    index_header.ulPacketLen = 36 + (18 * packets_c);
    index_header.ulDataLen = 12 + (18 * packets_c);
    index_header.ubyHdrVer = 0x06;
    index_header.ubySeqNum = seq;
    index_header.ubyPacketFlags = 0;
    index_header.ubyDataType = 0x03;
    index_header.uChecksum = 0;
    uint64_t sums = 0xeb25 + ((12 + (18 * packets_c)) * 2);
    sums += 24 + 6 + seq + 3;
    for (int i = 0; i <= 6; i++){
        sums += index_header.aubyRefTime[i];
    }
    sums &= 0xffff;
    index_header.uChecksum = (uint16_t)sums;
    fwrite(&index_header, 24, 1, output);

    // CSDW
    int32_t csdw = 0;
    csdw &= (1 << 31);
    csdw &= (1 << 30);
    fwrite(&csdw, 4, 1, output);

    // File size
    fwrite(&offset, 8, 1, output);

    // Packets
    for (int i = 0; i <= packets_c; i++){
        SuI106Ch10Header packet = packets[i];

        // IPTS
        fwrite(&index_header.aubyRefTime, 1, 6, output);

        // Index entry
        int8_t filler = 0;
        fwrite(&filler, 1, 1, output);
        fwrite(&packet.ubyDataType, 1, 1, output);
        fwrite(&packet.uChID, 2, 1, output);
        fwrite(&offsets[i], 8, 1, output);
    }
}

int increment(seq){
    seq++;
    if (seq > 0xFF){
        seq = 0;
    }
    return seq;
}

int main(int argc, char ** argv){
    DocoptArgs args = docopt(argc, argv, 1, "1");
    int input_handle;
    SuI106Ch10Header header;
    void * buffer = malloc(24);

    // Validate arguments and offer help.
    if (argc < 3){
        printf(args.usage_pattern);
        return quit(1);
    }

    // Open file for reading.
```

```
EnI106Status status = enI106Ch10Open(&input_handle, argv[1], I106_READ);
if (status != I106_OK){
    char msg[200] = "Error opening source file: ";
    strcat(msg, argv[1]);
    return error(msg);
}

// Open output file.
FILE * output = fopen(argv[2], "wb");
if (output == NULL){
    return error("Couldn't open destination file.");
}

int packets_c = 0;
SuI106Ch10Header packets_arr[20000];
int64_t offsets[20000];
SuI106Ch10Header last_packet;
int last_packet_at;
int node_seq = 0;
int64_t offset;
int *hdrptr;
int hdrlen;

// Parse loop.
while (1){

    enI106Ch10GetPos(input_handle, &offset);

    // Read next header or exit.
    status = enI106Ch10ReadNextHeader(input_handle, &header);
    if (status != I106_OK){
        printf("Finished");
        return quit(0);
    }

    // Ensure that we're interested in this particular packet.
    if (header.ubyDataType == 0x3){
        continue;
    }

    // Read packet data.
    buffer = realloc(buffer, header.ulPacketLen);
    status = enI106Ch10ReadDataFile(input_handle, header.ulPacketLen, buffer);
    if (status != I106_OK){
        printf("Error reading packet.");
        continue;
    }

    if (header.ubyDataType != 0x03){
        hdrptr = &header;
        if (header.ubyPacketFlags & (0x1 << 7)){
            hdrlen = 36;
        }
        else {
            hdrlen = 24;
        }

        // Write packet to file.
        fwrite(hdrptr, hdrlen, 1, output);
        fwrite(buffer, header.ulDataLen, 1, output);

        last_packet = header;
        //enI106Ch10GetPos(input_handle, &last_packet_at);
```

```
            packets_c++;
            packets_arr[packets_c] = header;
            offsets[packets_c] = offset - header.ulPacketLen;
        }

        if (args.strip){
            continue;
        }

        if ((header.ubyDataType == 0x02 || header.ubyDataType == 0x11) || (packets_c >
20000)){
            // Generate node packet.
            gen_node(offset, &packets_arr, packets_c, offsets, output, node_seq);
            node_seq = increment(node_seq);
            packets_c = 0;
        }

    }

    if (args.strip){
        printf("Stripped existing indices.");
    }
}
```

Appendix O

Example Python Program – Reindex Chapter 10 Files

The following software program opens a Chapter 10 file for reading and writes its content to another file while stripping and (optionally) rebuilding index packets. This sample uses the Python wrapper to the irig106lib C library.

```
#!/usr/bin/env python

"""
  reindex.py - Strip and (optionally) rebuild index packets for a Chapter 10
    file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: c10_reindex.py <src> <dst> [options]

Options:
    -s, --strip  Strip existing index packets and exit.
    -f, --force  Overwrite existing files."""

from array import array
from contextlib import closing
import os
import struct

from docopt import docopt

from Py106.Packet import IO, FileMode, Status
```

```
def gen_node(packets, seq=0):
    """Generate an index node packet."""

    # (rtc_low, rtc_high, channel_id, data_type, pos)

    print 'Index node for %s packets' % len(packets)

    packet = bytes()

    # Header
    values = [0xeb25,
              0,
              24 + 4 + (20 * len(packets)),
              4 + (20 * len(packets)),
              0x06,
              seq,
              0,
              0x03] + list(packets[-1][0])

    sums = struct.pack('HHIIBBBBBBBBBB', *values)
    sums = sum(array('H', sums)) & 0xffff
    values.append(sums)
    packet += struct.pack('HHIIBBBBBBBBBBH', *values)

    # CSDW
    csdw = 0x0000
    csdw &= 1 << 31
    csdw &= 1 << 30
    csdw += len(packets)
    packet += struct.pack('I', csdw)

    # File Length (at start of node)
    pos = packets[-1][-1] + packets[-1][3]
    packet += struct.pack('Q', pos)

    # Packets
    for p in packets:
        ipts = struct.pack('BBBBBB', *p[0])
        index = struct.pack('xHHQ', p[1], p[2], p[4])
        packet += ipts + index

    return pos, packet


def gen_root(nodes, last, seq, last_packet):
    """Generate a root index packet."""

    print 'Root index for: %s nodes' % len(nodes)

    packet = bytes()

    # Header
    values = [0xeb25,
              0,
              24 + 4 + 8 + 8 + (16 * len(packets)),
              4 + 8 + 8 + (16 * len(packets)),
              0x06,
              seq,
              0,
              0x03] + list(last_packet[0])

    sums = struct.pack('HHIIBBBBBBBBBB', *values)
```

```
    sums = sum(array('H', sums)) & 0xffff
    values.append(sums)
    packet += struct.pack('HHIIBBBBBBBBBBH', *values)

    # CSDW
    csdw = 0x0000
    csdw &= 1 << 30
    csdw += len(nodes)
    packet += struct.pack('I', csdw)

    # File Length (at start of node)
    pos = last_packet[-1] + last_packet[3]
    packet += struct.pack('Q', pos)

    # Node Packets
    for node in nodes:
        ipts = struct.pack('BBBBBB', *last_packet[0])
        offset = struct.pack('Q', pos - node)
        packet += ipts + offset

    if last is None:
        last = pos
    packet += struct.pack('Q', last)

    return pos, packet


def increment(seq):
    """Increment the sequence number or reset it."""

    seq += 1
    if seq > 0xFF:
        seq = 0
    return seq


if __name__ == '__main__':
    args = docopt(__doc__)

    # Don't overwrite unless explicitly required.
    if os.path.exists(args['<dst>']) and not args['--force']:
        print('dst file already exists. Use -f to overwrite.')
        raise SystemExit

    with open(args['<dst>'], 'wb') as out, closing(IO()) as io:

        status = io.open(args['<src>'], FileMode.READ)
        if status != Status.OK:
            print "Error opening data file %s" % args['<src>']
            raise SystemExit

        # Packets for indexing.
        packets, nodes = [], []
        node_seq = 0
        last_root = None
        last_packet = None

        while status == Status.OK:
            status = io.read_next_header()
            if io.read_data() != Status.OK:
                continue
            if io.Header.DataType == 0x03:
                continue
```

```
            header = buffer(io.Header)[:]
            out.write(header)
            raw = io.Buffer.raw[:io.Header.PacketLen - len(header)]
            out.write(raw)

            # Just stripping existing indices so move along.
            if args['--strip']:
                continue

            # (time, channel_id, data_type, length, pos)
            packet = (io.Header.RefTime, io.Header.Time[1],
                      io.Header.ChID, io.Header.DataType, io.Header.DataLen,
                      io.get_pos()[1] - io.Header.PacketLen)
            packets.append(packet)
            last_packet = packet

            # Projected index node packet size.
            size = 24 + 4 + (20 * len(packets))
            if io.Header.DataType in (0x02, 0x11) or size > 524000:
                offset, raw = gen_node(packets)
                nodes.append(offset)
                out.write(raw)
                packets = []

                # Projected root index packet size.
                size = 24 + 4 + (16 * len(nodes)) + 16
                if size > 524000:
                    last_root, raw = gen_root(nodes, last_root, node_seq,
                                              last_packet)
                    out.write(raw)
                    node_seq = increment(node_seq)
                    nodes = []

        # Final indices.
        if packets:
            offset, raw = gen_node(packets)
            nodes.append(offset)
            out.write(raw)
        if nodes:
            last_root, raw = gen_root(nodes, last_root, node_seq, last_packet)
            out.write(raw)

    if args['--strip']:
        print 'Stripped existing indices.'
```

Appendix P

Example Python Program – Reindex Chapter 10 Files

The following software program opens a Chapter 10 file for reading and writes its content to another file while stripping and (optionally) rebuilding index packets. This sample uses the PyChapter10 library.

```
#!/usr/bin/env python

"""
  reindex.py - Strip and (optionally) rebuild index packets for a Chapter 10
    file.

 Copyright (c) 2015 Micah Ferrill

 All rights reserved.

 Redistribution and use in source and binary forms, with or without
 modification, are permitted provided that the following conditions are
 met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in the
     documentation and/or other materials provided with the distribution.

   * Neither the name Irig106.org nor the names of its contributors may
     be used to endorse or promote products derived from this software
     without specific prior written permission.

 This software is provided by the copyright holders and contributors
 "as is" and any express or implied warranties, including, but not
 limited to, the implied warranties of merchantability and fitness for
 a particular purpose are disclaimed. In no event shall the copyright
 owner or contributors be liable for any direct, indirect, incidental,
 special, exemplary, or consequential damages (including, but not
 limited to, procurement of substitute goods or services; loss of use,
 data, or profits; or business interruption) however caused and on any
 theory of liability, whether in contract, strict liability, or tort
 (including negligence or otherwise) arising in any way out of the use
 of this software, even if advised of the possibility of such damage.
"""

__doc__ = """usage: reindex.py <src> <dst> [options]

Options:
    -s, --strip  Strip existing index packets and exit.
    -f, --force  Overwrite existing files."""

from array import array
import os
import struct

from docopt import docopt

from chapter10 import C10
from walk import walk_packets
```

```
def gen_node(packets, seq=0):
    """Generate an index node packet."""

    print 'Index node for %s packets' % len(packets)

    packet = bytes()

    # Header
    values = [0xeb25,
              0,
              24 + 4 + 8 + (20 * len(packets)),
              4 + 8 + (20 * len(packets)),
              0x06,
              seq,
              0,
              0x03,
              packets[-1].rtc_low,
              packets[-1].rtc_high]

    sums = struct.pack('HHIIBBBBIH', *values)
    sums = sum(array('H', sums)) & 0xffff
    values.append(sums)
    packet += struct.pack('HHIIBBBBIHH', *values)

    # CSDW
    csdw = 0x0000
    csdw &= 1 << 31
    csdw &= 1 << 30
    csdw += len(packets)
    packet += struct.pack('I', csdw)

    # File Length (at start of node)
    pos = packets[-1].pos + packets[-1].packet_length
    packet += struct.pack('Q', pos)

    # Packets
    for p in packets:
        ipts = struct.pack('HH', p.rtc_low & 0xffff, p.rtc_high & 0xffff)
        index = struct.pack('xHHQ', p.channel_id, p.data_type, p.pos)
        packet += ipts + index

    return pos, packet


def gen_root(nodes, last, seq, last_packet):
    """Generate a root index packet."""

    print 'Root index for: %s nodes' % len(nodes)

    packet = bytes()

    # Header
    values = [0xeb25,
              0,
              24 + 4 + 8 + 8 + (16 * len(packets)),
              4 + 8 + 8 + (16 * len(packets)),
              0x06,
              seq,
              0,
              0x03,
              last_packet.rtc_low,
              last_packet.rtc_high]
```

```
    sums = struct.pack('HHIIBBBBIH', *values)
    sums = sum(array('H', sums)) & 0xffff
    values.append(sums)
    packet += struct.pack('HHIIBBBBIHH', *values)

    # CSDW
    csdw = 0x0000
    csdw &= 1 << 30
    csdw += len(nodes)
    packet += struct.pack('I', csdw)

    # File Length (at start of node)
    pos = last_packet.pos + last_packet.packet_length
    packet += struct.pack('Q', pos)

    # Node Packets
    for node in nodes:
        ipts = struct.pack('HH', last_packet.rtc_low & 0xffff,
                           last_packet.rtc_high & 0xffff)
        offset = struct.pack('Q', pos - node)
        packet += ipts + offset

    if last is None:
        last = last_packet.pos + last_packet.packet_length
    packet += struct.pack('Q', last)

    return pos, packet


def increment(seq):
    """Increment the sequence number or reset it."""

    seq += 1
    if seq > 0xFF:
        seq = 0
    return seq


if __name__ == '__main__':
    args = docopt(__doc__)

    # Don't overwrite unless explicitly required.
    if os.path.exists(args['<dst>']) and not args['--force']:
        print('dst file already exists. Use -f to overwrite.')
        raise SystemExit

    with open(args['<dst>'], 'wb') as out:

        # Packets for indexing.
        packets, nodes = [], []
        node_seq = 0
        last_root = None
        last_packet = None

        for packet in walk_packets(C10(args['<src>']), args):
            last_packet = packet
            if packet.data_type == 0x03:
                continue

            raw = bytes(packet)
            if len(raw) == packet.packet_length:
                out.write(raw)
```

```
            # Just stripping existing indices so move along.
            if args['--strip']:
                continue

            packets.append(packet)

            # Projected index node packet size.
            size = 24 + 4 + 8 + (20 * len(packets))
            if packet.data_type in (0x02, 0x11) or size > 524000:
                offset, raw = gen_node(packets, node_seq)
                node_seq = increment(node_seq)
                nodes.append(offset)
                out.write(raw)
                packets = []

                # Projected root index packet size.
                size = 24 + 4 + (16 * len(nodes)) + 16
                if size > 524000:
                    last_root, raw = gen_root(nodes, last_root, node_seq,
                                              last_packet)
                    out.write(raw)
                    node_seq = increment(node_seq)
                    nodes = []

        # Final indices.
        if packets:
            offset, raw = gen_node(packets)
            nodes.append(offset)
            out.write(raw)
        if nodes:
            offset, raw = gen_root(nodes, last_root, node_seq, last_packet)
            out.write(raw)

    if args['--strip']:
        print('Stripped existing indices.')
```

Appendix Q

XML Mapping

Q.1 Introduction

This specification is a reply to the RCC change request CR031.

The intention of this specification is to have a way to define IRIG 106 Chapter 10 files for testing purposes in a readable XML format. This would allow tools to convert between XML and Chapter 10 back and forth. The focus of XML is readability and not memory efficiency or processing efficiency. Therefore the area of application for this specification is the generation of smaller test files (although nothing technically prevents having large files).

The conversion from XML to Chapter 10 opens these opportunities for example:

- Setup of test data for the verification of Chapter 10 analysis software;
- Preparation of exactly specified data sets for replay.

The conversion from Chapter 10 to XML opens these opportunities for example:

- Chapter 10 files (for example from recorders under development) could be converted to XML for closer inspection and to verify possible corrections (by converting the manually corrected XML file back to Chapter 10);
- Specific error cases can be created for test cases by converting real files to XML, injecting the error and converting back for replay.

Q.2 Changes

See the XSD file for a brief change history.

Q.3 Concepts

This specification provides an XML schema that defines many but not all restrictions for valid files. Having validated XML files is the precondition to any further conversion.

The XML files will basically be a sequence of Chapter 10 packets, each defined within one XML **Packet** tag.

Within one Packet tag, optional secondary headers and data type specific content can be placed.

Q.3.1 Raw data

Many places in the file allow the placement of raw data, for example for the creation of undefined packet types or to create errors. If anything cannot be defined in a structured way by the use of XML tags and attributes, it can still be defined by defining the raw data directly.

Q.3.2 Endianness

Chapter 10 files use little endian to encode attributes spanning over several bytes. Therefore a checksum attribute defined as 0x12345678 shall be stored in the byte order 78 56 34 12. If any of the XML attributes separates the data in groups however, then little endian byte order shall only be used within each group. For example if some raw data is defined as 16-bit

hex words like "1234 5678", then the data shall be stored in the order 34 12 78 56 because the values form two groups and little endian only applies within the group 1234 and the group 5678.

Q.3.3 Auto values

Often attributes offer "auto-values". This means that the user is not providing these values but the conversion tool is supposed to do this. Often there will be three options:

- Automatic calculation by conversion tool
- Automatic calculation by conversion tool, modified somehow by users choice (offset)
- Fixed value (user provided content)

An example would be a checksum value, for which following situations could occur

- The user omitted the value: The calculation will be automatic
- The user provided a relative value like +5 or −12: The result of the automatic calculation will be modified by +5 or −12 to create an invalid checksum.
- The user provided a fixed value: The fixed value will be used

Auto-values require an explicit definition of what they relate to. An auto-value that is calculating the number of MIL-STD 1553 messages within one packet for example requires explicit definition of the MIL-STD 1553 messages via XML tags. Definitions via a block of raw data will not count.

If you use a fixed value somewhere remember you need to update it manually when you change other data.

Q.3.4 Hexadecimal vs decimal

There are attributes that require the data to be defined in hexadecimal and others that require decimal data. Most attributes accept both. In this case, hexadecimal data shall be prefixed by "0x".

Q.3.5 Flag words

Data structures like flag words (like packet flags or CSDW) will be available to define as a whole or with its sub elements. In general the definition for the flag word will be used as a base that is then bitwise modified by the more specific sub elements within the flag word that are explicitly defined in XML.

Example:

PacketFlags="0xC4" RTCSyncError="True"

The packet flags of 0xC4 indicate that secondary headers are present and used as IEEE1588 intra-packet time. The RTCSyncError additionally sets the bit to indicate an RTC sync error so the stored flag word becomes 0xE4.

Q.3.6 RTC

Packet RTCs have several options to define which are described in the XSD file. Here are some examples.

1234	RTC = 1234
0xFF	RTC = 255
p+120	RTC is 120 ticks behind the previous packet
c+120	RTC is 120 ticks behind the previous packet on the same channel

Q.3.7 Channel defaults

You can define default values for packet flags, data type and data type version per channel. You can also change these defaults at different places in the file. You can override this in individual packets

Q.3.8 Various

All data that is not further specified shall be 0.

Some attributes just have one possible value. For example the RTCSyncError from the previous example can just be "true". Instead of setting it to false, the attribute must be omitted.

The XML schema (XSD file) contains some additional documentation for some elements.

Q.4 Specific data types

Q.4.1 TMATS

The TMATS data can be provided directly or as include file.

If you directly provide TMATS data you can use the ASCII tag. Note that you must make sure that the characters you use do not conflict with the XML syntax. Also as a result of the XML specification itself all line breaks will be converted to a single line feed (ASCII code 0x0A). Also XML prohibits the 0x00 ASCII code inside the data.

Q.4.2 Time

Time packets can be either defined relative to the last time packet or you can provide an absolute time definition.

Q.4.3 Real data types

Currently ARINC 429, MIL-STD 1553, UART, and CAN bus data can be defined in a structured way. Other data types can always be provided as raw data inside the packet.

Q.5 Sample files

There are seven sample files in XML and converted Chapter 10 format.

Arinc429/Mil1553/UART	These show how data for ARINC 429, MIL-STD-1553, and UART can be defined in a structured way.
CANBus	Demonstrates how data can be defined as raw data if the data type is not yet supported by XML. The same CAN bus data is defined as: • Raw data • Structured definition • Definition in DATaRec4 format (Zodiac Data systems CH10 extension)

NMEA0183	Shows an NMEA 0183 GPS definition using structured UART message definition.
RTCWrap	Shows how specific elements of packet decoding can be tested. Here the RTC wraps from the highest value to 0 and you can check how your CH10 software reacts on it (this is no unrealistic case since RTC has no defined starting value).
mappingFeatures	Shows many other features of the XML to CH10 mapping. The created file contains a lot of intentional errors like arbitrary data inserted between packets, checksum and sequence errors, etc. It also shows things like relative packet time definition and other concepts explained above. This sample uses an external TMATS file. You must download it and place it in the same directory or adapt the path.

Appendix R

XML Schema Definition (XSD)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/XMLCH10Mapping"
xmlns:cns="http://www.example.org/XMLCH10Mapping" elementFormDefault="qualified">
<!--  This proposed specification has the internal version number 0.3.1.
       History:
       Version 0.3.1
       - for automatic Arinc 429 parity handling changed within ARINC429MessageType:
         - renamed Data attribute to Raw32
         - added Raw31 attribute
         - added GenerateWrongParity attribute

       Version 0.3:
       - added IRIG 106-2015 version flags
       - added CAN Bus definition

       Version 0.2:
       - renamed all CRC to checksum
       - data CRC now requires 0x prefix
       - Raw time definition in secondary headers
 -->

       <xs:element name="ch10">
              <xs:complexType>
                    <xs:sequence minOccurs="0" maxOccurs="unbounded">
                          <xs:choice>
                                <xs:element name="Packet" type="cns:PacketType"/>
                                <xs:element name="ChannelDefaults"
type="cns:ChannelDefaultsType"/>
                                <xs:group ref="cns:RawDataGroup"/>
                          </xs:choice>
                    </xs:sequence>
             </xs:complexType>
       </xs:element>


<!-- Packet Header / Trailer -->
       <xs:complexType name="PacketType">
             <xs:sequence minOccurs="0">
                    <xs:element name="SecondaryHeader" type="cns:SecondaryHeaderType"
minOccurs="0"/>
                    <xs:choice minOccurs="0">
                          <xs:group ref="cns:RawDataGroup" minOccurs="0"
maxOccurs="unbounded" />
                          <xs:element name="UARTData" type="cns:UARTDataType"/>
                          <xs:element name="Mil1553Data" type="cns:Mil1553DataType"/>
                          <xs:element name="ARINC429Data"
type="cns:ARINC429DataType"/>
                          <xs:element name="CANBusData" type="cns:CANBusDataType"/>
                          <xs:element name="TMATSData" type="cns:TMATSDataType"/>
                          <xs:element name="TimeData" type="cns:TimeDataType"/>
                    </xs:choice>
             </xs:sequence>

             <!-- Header -->
             <xs:attribute name="ChannelID" type="cns:U16HexDec" use="required"/>
             <xs:attribute name="RTC" type="cns:PacketRTCType" use="required"/>
```

```
            <xs:attribute name="PacketSyncPattern" type="cns:U16HexDec"/>
            <xs:attribute name="PacketLength" type="cns:RelativeU32HexDec">
                  <xs:annotation><xs:documentation>If this attribute is not used,
the proper length is calculated. If it contains an unsigned number this is used as a
fixed value. Any signed numbers (including positive sign) will be used to manipulate
the proper length calculation. The number itself can start with 0x for hex
values</xs:documentation></xs:annotation>
            </xs:attribute>
            <xs:attribute name="DataLength" type="cns:RelativeU32HexDec">
                  <xs:annotation><xs:documentation>If this attribute is not used,
the proper length is calculated. If it contains an unsigned number this is used as a
fixed value. Any signed numbers (including positive sign) will be used to manipulate
the proper length calculation. The number itself can start with 0x for hex
values</xs:documentation></xs:annotation>
            </xs:attribute>

            <xs:attributeGroup ref="cns:dataTypeVersionDefinition"/>

            <xs:attribute name="SequenceNumber" type="cns:RelativeU8HexDec">
                  <xs:annotation><xs:documentation>If this attribute is not used,
the proper sequence number is calculated. If it contains an unsigned number this is
used as a fixed value. Any signed numbers (including positive sign) will be used to
manipulate the proper sequence calculation. The number itself can start with 0x for
hex values</xs:documentation></xs:annotation>
            </xs:attribute>

            <xs:attributeGroup ref="cns:packetFlagsDefinition"/>
            <xs:attributeGroup ref="cns:dataTypeDefinition"/>

            <xs:attribute name="HeaderChecksum" type="cns:RelativeU16HexDec">
                  <xs:annotation><xs:documentation>if absent automatic, otherwise
the hex value is used as checksum. If there is a sign in front (+/-) it is used as an
offset to the calculated checksum. The number itself can start with 0x for hex
values</xs:documentation></xs:annotation>
            </xs:attribute>

            <!-- Trailer -->
            <xs:attribute name="Filler" type="cns:HexBytesType">
                  <xs:annotation><xs:documentation>if absent automatic, otherwise
adds as many bytes as hex values are available</xs:documentation></xs:annotation>
            </xs:attribute>

            <xs:attribute name="DataChecksum" type="cns:ChecksumType">
                  <xs:annotation><xs:documentation>if absent automatic, otherwise
adds as many bytes as hex values are available (add leading 0 to identify number of
bytes). The data will be added in little endian order. values with a sign are
interpreted as offset for the automatic calculation and will use the number of bytes
defined in the packet flags</xs:documentation></xs:annotation>
            </xs:attribute>
      </xs:complexType>

      <xs:simpleType name="ChecksumType">
            <xs:restriction base="xs:string">
                  <xs:pattern value="[\+\-]?(([0-9]+)|(0x[0-9a-fA-
F]+))"></xs:pattern>
            </xs:restriction>
      </xs:simpleType>

      <xs:simpleType name="PacketRTCType">
            <xs:annotation>
                  <xs:documentation>
```

```
                        Absolute or relative RTC values. Absolute values are simple
unsigned integers. Values starting with p are relative to the last packet, values
starting with c are relative to previous packet on same channel. The second character
must then be a sign to indicate the direction of the following offset. Relative and
absolute values can start with 0x right before the first digit to indicate hex values
                        </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                        <xs:pattern value="((c\+)|(c\-)|(p\+)|(p\-))?(([0-9]{1,15})|(0x[0-
9a-fA-F]{1,12}))"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="DataTypeType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="UART Format 0"/>
                        <xs:enumeration value="1553 Format 1"/>
                        <xs:enumeration value="ARINC-429 Format 0"/>
                        <xs:enumeration value="Time Format 1"/>
                        <xs:enumeration value="TMATS"/>
                        <xs:enumeration value="CAN Bus"/>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="DataTypeVersionType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="106-04"/>
                        <xs:enumeration value="106-05"/>
                        <xs:enumeration value="106-07"/>
                        <xs:enumeration value="106-09"/>
                        <xs:enumeration value="106-11"/>
                        <xs:enumeration value="106-13"/>
                        <xs:enumeration value="106-15"/>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="SecondaryHeaderTimeFormatType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Chapter 4 Binary"/>
                        <xs:enumeration value="IEEE 1588"/>
                        <xs:enumeration value="ERTC"/>
                        <xs:enumeration value="Reserved"/>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="ChecksumTypeType">
                <xs:restriction base="xs:string">
                        <xs:enumeration value="Checksum0"/>
                        <xs:enumeration value="Checksum8"/>
                        <xs:enumeration value="Checksum16"/>
                        <xs:enumeration value="Checksum32"/>
                </xs:restriction>
        </xs:simpleType>


<!--  Secondary headers -->
        <xs:complexType name="SecondaryHeaderType">
                <xs:attributeGroup ref="cns:secondaryHeaderTimeDefinition"/>
                <xs:attribute name="Filler" type="cns:HexBytesType"/>
                <xs:attribute name="Checksum" type="cns:RelativeU16HexDec"/>
        </xs:complexType>

        <!-- Time packets -->
        <xs:complexType name="TimeDataType">
                <xs:sequence minOccurs="0" maxOccurs="unbounded">
                        <xs:choice>
```

```
                        <xs:element name="TimeDataRelativeContent"
type="cns:TimeDataRelativeContentType" />
                        <xs:element name="TimeDataContent"
type="cns:TimeDataContentType" />
                        <xs:group ref="cns:RawDataGroup" />
                  </xs:choice>
            </xs:sequence>
            <xs:attribute name="MonthYearAvailable" type="cns:TrueType" />
            <xs:attribute name="LeapYear" type="cns:TrueType" />
            <xs:attribute name="TimeFormat" type="cns:TimeDataTimeFormatType" />
            <xs:attribute name="TimeSource" type="cns:TimeDataTimeSourceType" />
            <xs:attribute name="CSDW" type="cns:U32HexDec" />
      </xs:complexType>

      <xs:complexType name="TimeDataContentType">
            <xs:attribute name="msec" type="xs:unsignedShort" use="optional" />
            <xs:attribute name="sec" type="xs:unsignedByte" use="required" />
            <xs:attribute name="min" type="xs:unsignedByte" use="required" />
            <xs:attribute name="hrs" type="xs:unsignedByte" use="required" />
            <xs:attribute name="day" type="xs:unsignedShort" use="required" />
            <xs:attribute name="mth" type="xs:unsignedByte" use="optional" />
            <xs:attribute name="year" type="xs:unsignedShort" use="optional" />
      </xs:complexType>

      <xs:complexType name="TimeDataRelativeContentType">
            <xs:attribute name="OffsetMs" type="xs:long" use="required">
                  <xs:annotation>
                        <xs:documentation>
                              Offset to last properly defined time packet on same
channel in milliseconds.
                              This may wrap the year respecting the LeapYear
attribute definition or CSDW flag of the last properly defined time packet
                              on the same channel. Even if the year is wrapping the
leap year attribute shall not be altered for this packet but use defined state.
                         </xs:documentation>
                  </xs:annotation>
            </xs:attribute>
      </xs:complexType>

      <xs:simpleType name="TimeDataTimeFormatType">
            <xs:restriction base="xs:string">
                  <xs:enumeration value="IRIG B" />
                  <xs:enumeration value="IRIG A" />
                  <xs:enumeration value="IRIG G" />
                  <xs:enumeration value="Internal" />
                  <xs:enumeration value="UTC Time From GPS" />
                  <xs:enumeration value="Native GPS Time" />
                  <xs:enumeration value="None" />
            </xs:restriction>
      </xs:simpleType>

      <xs:simpleType name="TimeDataTimeSourceType">
            <xs:restriction base="xs:string">
                  <xs:enumeration value="Internal" />
                  <xs:enumeration value="External" />
                  <xs:enumeration value="Internal From RMM" />
                  <xs:enumeration value="None" />
            </xs:restriction>
      </xs:simpleType>

      <xs:complexType name="ChannelDefaultsType">
            <xs:attribute name="ChannelID" type="cns:U16HexDec" use="required"/>
            <xs:attributeGroup ref="cns:packetFlagsDefinition"/>
```

```
		<xs:attributeGroup ref="cns:dataTypeDefinition"/>
		<xs:attributeGroup ref="cns:dataTypeVersionDefinition"/>
	</xs:complexType>


	<!-- TMATS packets -->
	<xs:complexType name="TMATSDataType">
		<xs:sequence minOccurs="0" maxOccurs="unbounded">
			<xs:choice>
				<xs:element name="IncludeFile" type="xs:string">
					<xs:annotation>
					<xs:documentation>If the filename is relative or has
no path at all it should be relative to the XML file
					</xs:documentation>
					</xs:annotation>
				</xs:element>
				<xs:group ref="cns:RawDataGroup" />
			</xs:choice>
		</xs:sequence>
		<xs:attribute name="XML" type="cns:TrueType" />
		<xs:attribute name="ConfigChange" type="cns:TrueType" />
		<xs:attribute name="Ch10Version" type="cns:TMATSDataCh10VersionType" />
		<xs:attribute name="CSDW" type="cns:U32HexDec" />
	</xs:complexType>

	<xs:simpleType name="TMATSDataCh10VersionType">
		<xs:restriction base="xs:string">
			<xs:enumeration value="106-07" />
			<xs:enumeration value="106-09" />
			<xs:enumeration value="106-11" />
			<xs:enumeration value="106-13" />
			<xs:enumeration value="106-15" />
		</xs:restriction>
	</xs:simpleType>


<!-- CANBus Packets -->
	<xs:complexType name="CANBusDataType">
		<xs:sequence minOccurs="0" maxOccurs="unbounded">
			<xs:choice>
				<xs:element name="CANBusMessage"
type="cns:CANBusMessageType" />
				<xs:group ref="cns:RawDataGroup" />
			</xs:choice>
		</xs:sequence>
		<xs:attribute name="MessageCount" type="cns:RelativeU24HexDec" />
		<xs:attribute name="CSDW" type="cns:U32HexDec" />
	</xs:complexType>

	<xs:complexType name="CANBusMessageType">
		<xs:sequence minOccurs="0" maxOccurs="unbounded">
			<xs:group ref="cns:RawDataGroup" />
		</xs:sequence>

		<xs:attribute name="RTC" type="cns:RelativeU64HexDec">
			<xs:annotation>
				<xs:documentation>This is used as a 64 bit RTC value, i.e.
the first
					16 bits are normally filled with 0. If RTC attribute
is used, it
					shall take precedence over date/time. If this has a
sign (+/-),
					then this is interpreted relative to the packet RTC
```

```
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>

            <xs:attributeGroup ref="cns:secondaryHeaderTimeDefinition" />

            <xs:attribute name="IPMessageHeader" type="cns:U32HexDec" />
            <xs:attribute name="IPIDWord" type="cns:U32HexDecNone" />
            <xs:attribute name="DataError" type="cns:TrueType" />
            <xs:attribute name="FormatError" type="cns:TrueType" />
            <xs:attribute name="ExtendedIdentifier" type="cns:TrueType" />
            <xs:attribute name="RemoteTransferRequest" type="cns:TrueType" />
            <xs:attribute name="CANBusID" type="cns:U32HexDec" />
            <xs:attribute name="Subchannel" type="cns:U8HexDec" />
            <xs:attribute name="LengthBytes" type="cns:RelativeU8HexDec" >
            <xs:annotation>
                        <xs:documentation>
                              The length attribute includes the 4 bytes of the
Intra packet ID word per CH10.
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>
      </xs:complexType>


      <!-- Mil1553 Packets -->
      <xs:complexType name="Mil1553DataType">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:choice>
                        <xs:element name="Mil1553Message"
type="cns:Mil1553MessageType" />
                        <xs:group ref="cns:RawDataGroup" />
                  </xs:choice>
            </xs:sequence>
            <xs:attribute name="MessageCount" type="cns:RelativeU24HexDec" />
            <xs:attribute name="TimeTagBits" type="cns:Mil1553TimeTagBitsType" />
            <xs:attribute name="CSDW" type="cns:U32HexDec" />
      </xs:complexType>

      <xs:complexType name="Mil1553MessageType">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:group ref="cns:RawDataGroup" />
            </xs:sequence>

            <xs:attribute name="RTC" type="cns:RelativeU64HexDec">
                  <xs:annotation>
                        <xs:documentation>This is used as a 64 bit RTC value, i.e.
the first
                              16 bits are normally filled with 0. If RTC attribute
is used, it
                              shall take precedence over date/time. If this has a
sign (+/-),
                              then this is interpreted relative to the packet RTC
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>

            <xs:attributeGroup ref="cns:secondaryHeaderTimeDefinition" />

            <xs:attribute name="BlockStatusWord" type="cns:U16HexDec" />
            <xs:attribute name="BusB" type="cns:TrueType" />
            <xs:attribute name="MessageError" type="cns:TrueType" />
            <xs:attribute name="RTRTTransfer" type="cns:TrueType" />
```

```
            <xs:attribute name="FormatError" type="cns:TrueType" />
            <xs:attribute name="ResponseTimeOut" type="cns:TrueType" />
            <xs:attribute name="WordCountError" type="cns:TrueType" />
            <xs:attribute name="SyncTypeError" type="cns:TrueType" />
            <xs:attribute name="InvalidWordError" type="cns:TrueType" />
            <xs:attribute name="GapTime1" type="cns:U8HexDec" />
            <xs:attribute name="GapTime2" type="cns:U8HexDec" />
            <xs:attribute name="LengthBytes" type="cns:RelativeU16HexDec" /><!-- TODO
check if this can be less than 16 bits -->
      </xs:complexType>

      <xs:simpleType name="Mil1553TimeTagBitsType">
            <xs:restriction base="xs:string">
                  <xs:enumeration value="Last Bit Of Last Word" />
                  <xs:enumeration value="First Bit Of First Word" />
                  <xs:enumeration value="Last Bit Of First Command" />
            </xs:restriction>
      </xs:simpleType>


      <!-- ARINC429 Packets -->
      <xs:complexType name="ARINC429DataType">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:choice>
                        <xs:element name="ARINC429Message"
type="cns:ARINC429MessageType" />
                        <xs:group ref="cns:RawDataGroup" />
                  </xs:choice>
            </xs:sequence>
            <xs:attribute name="MessageCount" type="cns:RelativeU16HexDec" />
            <xs:attribute name="CSDW" type="cns:U32HexDec" />
      </xs:complexType>

      <xs:complexType name="ARINC429MessageType">
            <xs:attribute name="IPDH" type="cns:U32HexDec" />
            <xs:attribute name="Subchannel" type="cns:U8HexDec" />
            <xs:attribute name="FormatError" type="cns:TrueType" />
            <xs:attribute name="ParityError" type="cns:TrueType" />
            <xs:attribute name="HighSpeed" type="cns:TrueType" />
            <xs:attribute name="GapTime" type="xs:unsignedInt" />
            <xs:attribute name="Raw32" type="cns:U32HexDec">
                  <xs:annotation>
                        <xs:documentation>The raw bits including the parity.
                              If the Raw31 attribute is given, Raw32 shall be
ignored.
                              The state of the ParityError flag is not influenced.
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>
            <xs:attribute name="Raw31" type="cns:U32HexDec">
                  <xs:annotation>
                        <xs:documentation>The raw bits excluding the parity.
                              The parity bit for the raw bits is generated
automatically to be correct unless the GenerateWrongParity attribute is set in which
case the parity bit is generated incorrect.
                              If the Raw31 attribute is given, Raw32 shall be
ignored.
                              The state of the ParityError flag is not influenced.
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>
            <xs:attribute name="GenerateWrongParity" type="cns:TrueType">
                  <xs:annotation>
```

```
                        <xs:documentation>If this is set a wrong parity bit is
generated to the raw bits defined by Raw31.
                                If not correct parity is generated.
                                If Raw31 is not provided, there is no automatic
handling of the parity bit.
                                The state of the ParityError flag is not influenced.
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>
      </xs:complexType>



<!-- UART Packets -->
      <xs:complexType name="UARTDataType">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:choice>
                        <xs:element name="UARTMessage" type="cns:UARTMessageType"
/>
                        <xs:group ref="cns:RawDataGroup" />
                  </xs:choice>
            </xs:sequence>
            <xs:attribute name="IPTS" type="cns:TrueType" />
            <xs:attribute name="CSDW" type="cns:U32HexDec" />
      </xs:complexType>

      <xs:complexType name="UARTMessageType">
            <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:group ref="cns:RawDataGroup" />
            </xs:sequence>

            <xs:attribute name="RTC" type="cns:RelativeU64HexDec">
                  <xs:annotation>
                        <xs:documentation>This is used as a 64 bit RTC value, i.e.
the first
                                16 bits are normally filled with 0. If RTC attribute
is used, it
                                shall take precedence over date/time. If this has a
sign (+/-),
                                then this is interpreted relative to the packet RTC.
If neither RawTime not RTC nor date/time is used, the IPH is left out.
                        </xs:documentation>
                  </xs:annotation>
            </xs:attribute>

            <xs:attributeGroup ref="cns:secondaryHeaderTimeDefinition" />

            <xs:attribute name="IDWord" type="cns:U32HexDec" />
            <xs:attribute name="Subchannel" type="cns:U16HexDec" />
            <xs:attribute name="ParityError" type="cns:TrueType" />
            <xs:attribute name="LengthBytes" type="cns:RelativeU16HexDec" />
            <!--  TODO filler byte corruption settings -->
      </xs:complexType>



<!--  common type definitions -->
      <xs:group name="RawDataGroup">
            <xs:choice>
                  <xs:element name="HexBytes" type="cns:HexBytesType" />
                  <xs:element name="HexWords" type="cns:HexWordsType" />
                  <xs:element name="HexLongWords" type="cns:HexLongWordsType" />
                  <xs:element name="ASCII" type="cns:ASCIIStringType" />
```

```
            </xs:choice>
      </xs:group>

      <xs:attributeGroup name="dataTypeDefinition">
            <xs:attribute name="DataTypeRaw" type="cns:U8HexDec"/>
            <xs:attribute name="DataType" type="cns:DataTypeType">
                  <xs:annotation><xs:documentation>This takes precedence over
data_type_raw.</xs:documentation></xs:annotation>
            </xs:attribute>
      </xs:attributeGroup>

      <xs:attributeGroup name="dataTypeVersionDefinition">
            <xs:attribute name="DataTypeVersionRaw" type="cns:U8HexDec"/>
            <xs:attribute name="DataTypeVersion" type="cns:DataTypeVersionType">
                  <xs:annotation><xs:documentation>This takes precedence over
data_type_version_raw.</xs:documentation></xs:annotation>
            </xs:attribute>
      </xs:attributeGroup>



      <xs:attributeGroup name="packetFlagsDefinition">
            <xs:attribute name="PacketFlags" type="cns:U8HexDec"/>
            <xs:attribute name="SecondaryHeaderPresent" type="cns:TrueType"/>
            <xs:attribute name="SecondaryHeaderTimeForIntraPacketTime"
type="cns:TrueType"/>
            <xs:attribute name="RTCSyncError" type="cns:TrueType"/>
            <xs:attribute name="DataOverflow" type="cns:TrueType"/>
            <xs:attribute name="SecondaryHeaderTimeFormat"
type="cns:SecondaryHeaderTimeFormatType"/>
            <xs:attribute name="ChecksumType" type="cns:ChecksumTypeType"/>
      </xs:attributeGroup>

      <xs:attributeGroup name="secondaryHeaderTimeDefinition">
            <xs:attribute name="nsec" type="xs:unsignedShort" use="optional" />
            <xs:attribute name="usec" type="xs:unsignedShort" use="optional" />
            <xs:attribute name="msec" type="xs:unsignedShort" use="optional" />
            <xs:attribute name="sec" type="xs:unsignedByte" use="optional" />
            <xs:attribute name="min" type="xs:unsignedByte" use="optional" />
            <xs:attribute name="hrs" type="xs:unsignedByte" use="optional" />
            <xs:attribute name="day" type="xs:unsignedByte" use="optional" />
            <xs:attribute name="mth" type="xs:unsignedByte" use="optional">
                  <xs:annotation>
                        <xs:documentation>If this attribute is used and no ERTC or
RawTime, a IEEE-1588 time is created</xs:documentation>
                  </xs:annotation>
            </xs:attribute>
            <xs:attribute name="year" type="xs:unsignedShort" use="optional">
                  <xs:annotation>
                        <xs:documentation>If this attribute is used and no ERTC or
RawTime, a IEEE-1588 time is created</xs:documentation>
                  </xs:annotation>
            </xs:attribute>
            <xs:attribute name="ERTC" type="xs:unsignedLong">
                  <xs:annotation>
                        <xs:documentation>If this attribute is used it takes
precedence over
                              date/time but not RawTime. Any signed numbers
(including positive sign) will be
                              relative to the packet RTC (after extending the
packet RTC to 1ns
                              units). Numbers without sign are treated absolute
                        </xs:documentation>
```

```
                        </xs:annotation>
                </xs:attribute>
                <xs:attribute name="RawTime" type="cns:U64Hex">
                        <xs:annotation>
                                <xs:documentation>If this attribute is used it takes
precedence over
                                        date/time and ERTC. It will place the defined
hexadecimal value into the 8 time bytes (little endian)
                                </xs:documentation>
                        </xs:annotation>
                </xs:attribute>
        </xs:attributeGroup>


        <xs:simpleType name="ASCIIStringType">
                <xs:annotation>
                        <xs:documentation>
                                This represents a 7-Bit ASCII encoded string
                        </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                         <xs:pattern value="\p{IsBasicLatin}*"/>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="HexBytesType">
                <xs:annotation>
                        <xs:documentation>
                                This type represents a sequence of 8 bit hexadecimal values
separated by spaces. The values are stored in this sequence in the CH10 file. The
sequence may be empty.
                        </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                        <xs:pattern value="([a-fA-F0-9]{2}( [a-fA-F0-
9]{2})*)|"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="HexWordsType">
                <xs:annotation>
                        <xs:documentation>
                                This type represents a sequence of 16 bit hexadecimal
values separated by spaces. The values are stored in this sequence in the CH10 file.
When storing to the little endian CH10 file each value will be byte reversed. The
sequence may be empty.
                        </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                        <xs:pattern value="([a-fA-F0-9]{4}( [a-fA-F0-
9]{4})*)|"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="HexLongWordsType">
                <xs:annotation>
                        <xs:documentation>
                                This type represents a sequence of 32 bit hexadecimal
values separated by spaces. The values are stored in this sequence in the CH10 file.
When storing to the little endian CH10 file each value will be byte reversed. The
sequence may be empty.
                        </xs:documentation>
                </xs:annotation>
                <xs:restriction base="xs:string">
                        <xs:pattern value="([a-fA-F0-9]{8}( [a-fA-F0-
9]{8})*)|"></xs:pattern>
```

```
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="TrueType">
		<xs:annotation>
			<xs:documentation>
				This type is used for attributes that just have two states.
The false-state is indicated by omitting the attribute at all.
			</xs:documentation>
		</xs:annotation>
		<xs:restriction base="xs:string">
			 <xs:enumeration value="True"/>
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="NoneType">
		<xs:restriction base="xs:string">
			 <xs:enumeration value="None"/>
		</xs:restriction>
	</xs:simpleType>


	<xs:simpleType name="RelativeU8HexDec">
			<xs:annotation>
			<xs:documentation>
				This type represents a 8 bit unsigned integer that is
interpreted hexadecimal if the number is prefixed with 0x and decimal otherwise. The
first character is an optional + or - to indicate that the number shall be interpreted
relative to something (likely a calculated auto value).
			</xs:documentation>
		</xs:annotation>

		<xs:restriction base="xs:string">
			<xs:pattern value="[\+\-]?(([0-9]{1,3})|(0x[a-fA-F0-
9]{1,2}))"></xs:pattern>
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="RelativeU16HexDec">
			<xs:annotation>
			<xs:documentation>
				This type represents a 16 bit unsigned integer that is
interpreted hexadecimal if the number is prefixed with 0x and decimal otherwise. The
first character is an optional + or - to indicate that the number shall be interpreted
relative to something (likely a calculated auto value).
			</xs:documentation>
		</xs:annotation>
		<xs:restriction base="xs:string">
			<xs:pattern value="[\+\-]?(([0-9]{1,5})|(0x[a-fA-F0-
9]{1,4}))"></xs:pattern>
		</xs:restriction>
	</xs:simpleType>

	<xs:simpleType name="RelativeU24HexDec">
			<xs:annotation>
			<xs:documentation>
				This type represents a 24 bit unsigned integer that is
interpreted hexadecimal if the number is prefixed with 0x and decimal otherwise. The
first character is an optional + or - to indicate that the number shall be interpreted
relative to something (likely a calculated auto value).
			</xs:documentation>
		</xs:annotation>
		<xs:restriction base="xs:string">
```

```
                    <xs:pattern value="[\+\-]?(([0-9]{1,8})|(0x[a-fA-F0-
9]{1,6}))"></xs:pattern>
              </xs:restriction>
       </xs:simpleType>

       <xs:simpleType name="RelativeU32HexDec">
              <xs:annotation>
                    <xs:documentation>
                           This type represents a 32 bit unsigned integer that is
interpreted hexadecimal if the number is prefixed with 0x and decimal otherwise. The
first character is an optional + or - to indicate that the number shall be interpreted
relative to something (likely a calculated auto value).
                    </xs:documentation>
              </xs:annotation>
              <xs:restriction base="xs:string">
                    <xs:pattern value="[\+\-]?(([0-9]{1,10})|(0x[a-fA-F0-
9]{1,8}))"></xs:pattern>
              </xs:restriction>
       </xs:simpleType>

       <xs:simpleType name="RelativeU64HexDec">
              <xs:annotation>
                    <xs:documentation>
                           This type represents a 64 bit unsigned integer that is
interpreted hexadecimal if the number is prefixed with 0x and decimal otherwise. The
first character is an optional + or - to indicate that the number shall be interpreted
relative to something (likely a calculated auto value).
                    </xs:documentation>
              </xs:annotation>
              <xs:restriction base="xs:string">
                    <xs:pattern value="[\+\-]?(([0-9]{1,20})|(0x[a-fA-F0-
9]{1,16}))"></xs:pattern>
              </xs:restriction>
       </xs:simpleType>

       <xs:simpleType name="U8HexDec">
              <xs:annotation>
                    <xs:documentation>
                           This type represents a 8 bit unsigned integer that is
interpreted hexadecimal if it is prefixed with 0x and decimal otherwise.
                    </xs:documentation>
              </xs:annotation>
              <xs:union memberTypes="xs:unsignedByte cns:U8Hex" />
       </xs:simpleType>

       <xs:simpleType name="U8Hex">
              <xs:restriction base="xs:string">
                    <xs:pattern value="0x[a-fA-F0-9]{1,2}"></xs:pattern>
              </xs:restriction>
       </xs:simpleType>

       <xs:simpleType name="U16HexDec">
              <xs:annotation>
                    <xs:documentation>
                           This type represents a 16 bit unsigned integer that is
interpreted hexadecimal if it is prefixed with 0x and decimal otherwise.
                    </xs:documentation>
              </xs:annotation>
              <xs:union memberTypes="xs:unsignedShort cns:U16Hex" />
       </xs:simpleType>

       <xs:simpleType name="U16Hex">
              <xs:restriction base="xs:string">
```

```
                        <xs:pattern value="0x[a-fA-F0-9]{1,4}"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="U32HexDec">
                <xs:annotation>
                        <xs:documentation>
                                This type represents a 32 bit unsigned integer that is
interpreted hexadecimal if it is prefixed with 0x and decimal otherwise.
                        </xs:documentation>
                </xs:annotation>
                <xs:union memberTypes="xs:unsignedInt cns:U32Hex" />
        </xs:simpleType>

        <xs:simpleType name="U32Hex">
                <xs:restriction base="xs:string">
                        <xs:pattern value="0x[a-fA-F0-9]{1,8}"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="U64HexDec">
                <xs:annotation>
                        <xs:documentation>
                                This type represents a 64 bit unsigned integer that is
interpreted hexadecimal if it is prefixed with 0x and decimal otherwise.
                        </xs:documentation>
                </xs:annotation>
                <xs:union memberTypes="xs:unsignedLong cns:U64Hex" />
        </xs:simpleType>

        <xs:simpleType name="U64Hex">
                <xs:restriction base="xs:string">
                        <xs:pattern value="0x[a-fA-F0-9]{1,16}"></xs:pattern>
                </xs:restriction>
        </xs:simpleType>

        <xs:simpleType name="U32HexDecNone">
                <xs:annotation>
                        <xs:documentation>
                                This type allows the definitions like in U32HexDec but adds
the additional state "None" that is used to completely leave the 32 bits out
                        </xs:documentation>
                </xs:annotation>
                <xs:union memberTypes="cns:NoneType cns:U32HexDec" />
        </xs:simpleType>

</xs:schema>
```

This page intentionally left blank.

Appendix S

Citations

Aeronautical Radio, Inc. *Mark 33 Digital Information Transfer System (DITS)*. ARINC 429. Annapolis: ARINC, 1995.

European Air Group. "European Air Group Interface Control Document for Post Mission Interoperability." DF29125 Draft A Issue 01. July 2004. Retrieved 13 August 2019. Available to RCC members with Private Portal access at https://www.trmc.osd.mil/wiki/download/attachments/114820287/DF29125.pdf

Institute of Electrical and Electronics Engineers. *IEEE Standard for a High-Performance Serial Bus*. IEEE 1394-2008. New York: Institute of Electrical and Electronics Engineers, 2008.

International Organization for Standardization/International Electrotechnical Commission. "General sequential and progressive syntax", Annex B, section B.2, in *Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines*. ISO/IEC 10918-1:1994. May be superseded by update. Geneva: International Organization for Standardization, 1994.

———. *Information Technology – Generic Coding of Moving Pictures and Associated Audio Information: Systems*. ISO/IEC 13818-1:2019. June 2019. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/75928.html.

———. *Information technology -- JPEG 2000 image coding system: motion JPEG 2000*. ISO/IEC 15444-3:2007. Geneva: International Organization for Standardization, 2007.

———. *Information Technology – Small Computer System Interface (SCSI) Part 322: SCSI Block Commands-2 (SBC-2)*. ISO/IEC 14776-322:2007. February 2007. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/42101.html..

———. *Information Technology – Small Computer System Interface (SCSI) Part 412: Architecture Model-2 (SAM-2)*. ISO/IEC 14776-412. October 2006. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/39517.html.

———. *Information Technology – Small Computer System Interface (SCSI) Part 452: SCSI Primary Commands-2 (SPC-2)*. ISO/IEC 14776-452:2005. August 2005. Retrieved 12 August 2019. Available for purchase at https://www.iso.org/standard/38775.html.

International Telecommunications Union Telecommunication Standardization Sector. "Advanced Video Coding for Generic Audiovisual Services." ITU-T H.264. June 2019. May be superseded by update. Retrieved 13 August 2019. Available at https://www.itu.int/rec/T-REC-H.264.

Internet Engineering Task Force. "Administratively Scoped IP Multicast." RFC 2365. July 1998. May be superseded by update. Retrieved 13 August 2019. Available at https://datatracker.ietf.org/doc/rfc2365/.

———. "IANA Guidelines for IPv4 Multicast Address Assignments." RFC 3171. August 2001. Obsoleted by RFC 5771. Retrieved 13 August 2019. Available at https://datatracker.ietf.org/doc/rfc3171/.

———. "Internet Storage Name Service (iSNS)." RFC 4171. September 2005. May be superseded by update. Retrieved 12 August 2019. Available at https://datatracker.ietf.org/doc/rfc4171/

———. "Multi-Protocol Label Switching (MPLS) Support of Differentiated Services." RFC 3270. May 2002. Updated by RFC 5462. Retrieved 12 August 2019. Available at http://datatracker.ietf.org/doc/rfc3270/.

———. "Service Location Protocol, Version 2." RFC 2608. June 1999. Updated by RFC 3224. Retrieved 12 August 2019. Available at http://datatracker.ietf.org/doc/rfc2608/.

———. "Telnet Linemode Option." RFC 1184. March 2013. May be superseded by update. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc1184/.

———. "Telnet Option Specifications." RFC 855. May 1983. May be superseded by update. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc855/.

———. "Telnet Protocol Specification." RFC 854. May 1983. Updated by RFC 5198. Retrieved 26 June 2019. Available at http://datatracker.ietf.org/doc/rfc854/.

Lockheed Martin Corporation. "Advanced Weapons Multiplex Data Bus." 8 June 2010. May be superseded by update. Retrieved 3 June 2015. Available to RCC members with Private Portal access at https://www.trmc.osd.mil/wiki/download/attachments/114820295/16PP362B.pdf.

National Marine Electronics Association. "NMEA 0183 Interface Standard." V 4.11. November 2018. May be superseded by update. Retrieved 13 August 2019. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.

———. "NMEA 0183-HS Interface Standard." V 1.01. n.d. May be superseded by update. Retrieved 10 August 2016. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_0183_Standard.

———. "Standard for Serial-Data Networking of Marine Electronic Devices." Edition 3.101. March 2016. May be superseded by update. Retrieved 13 August 2019. Available for purchase at https://www.nmea.org/content/STANDARDS/NMEA_2000.

Radio Technical Commission for Maritime Services. "Standard for Networked Transport of RTCM via Internet Protocol (Ntrip)." RTCM 10410.1. Version 2.0 Amendment 1. June 2011. May be superseded by update. Retrieved 13 August 2019. Available for purchase at http://www.rtcm.org/differential-global-navigation-satellite--dgnss--standards.html.

Range Commanders Council. "Digital Data Bus Acquisition Formatting Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 8. July 2020. May be superseded by update. Retrieved

11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/chapter8.pdf.

———. "Digital Recording Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 10. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/chapter10.pdf.

———. "Pulse Code Modulation Standards" in *Telemetry Standards*. IRIG 106-20 Chapter 4. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/chapter4.pdf.

———. "Pulse Code Modulation Standards (Additional Information and Recommendations)" in *Telemetry Standards*. IRIG 106-20 Chapter 4 Appendix A. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/chapter4.pdf.

———. "Recorder & Reproducer Command and Control" in *Telemetry Standards*. IRIG 106-20 Chapter 6. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/chapter6.pdf.

———. *Telemetry Applications Handbook*. RCC 119-06. May 2006. May be superseded by update. Retrieved 12 August 2019. Available at https://www.trmc.osd.mil/wiki/download/attachments/91391305/119-06.pdf.

———. "Telemetry Attributes Transfer Standard" in *Telemetry Standards*. IRIG 106-20 Chapter 9. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/Chapter9.pdf.

———. *Telemetry Standards*. IRIG 106-20. July 2020. May be superseded by update. Retrieved 11 August 2020. Available at https://www.trmc.osd.mil/wiki/download/attachments/83068101/106-20_Telemetry_Standards.pdf.

Range Instrumentation System Program Office, Air Armament Center. "Interface Specification for the USAF Air Combat Test and Training System (ACTTS) Air-to-Ground, Air-to-Air, Ground-to-Air Data Links, and AIS Recording Formats." WMSP 98-01, Rev A, Chg 1. 19 May 2003. Retrieved 3 June 2015. Available to RCC members with Private Portal access at https://wsdmext.wsmr.army.mil/site/rccpri/Limited_Distribution_References/WMSP_98-01.doc.

Appendix T

References

Department of Defense. "Digital Time Division Command/Response Multiplex Data Bus." MIL-STD-1553C. 28 February 2018. May be superseded by update. Retrieved 13 August 2019. Available at https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=36973.

———. "National Imagery Transmission Format Version 2.1 for the National Imagery Transmission Format Standard." MIL-STD-2500C Change 2. 2 January 2019. May be superseded by update. Retrieved 13 August 2019. Available at https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=112606.

Institute of Electrical and Electronics Engineers. *IEEE Standard for Ethernet – Amendment 2: Media Access Control Parameters, Physical Layers, and Management Parameters for 25Gb/s Operation*. IEEE 802.3by-2016. September 2016. May be superseded by update. Retrieved 13 August 2019. Available for purchase at https://standards.ieee.org/findstds/standard/802.3by-2016.html.

———. *Information technology -- Generic coding of moving pictures and associated audio information -- Part 2: Video*. ISO/IEC 13818:2-2013. October 2013. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/61152.html.

———. *Information technology -- Generic coding of moving pictures and associated audio information -- Part 3: Audio*. ISO/IEC 13818:3-1998. April 1998. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/26797.html.

———. *Information technology -- Microprocessor systems -- Control and Status Registers (CSR) Architecture for microcomputer buses*. ISO/IEC 13213:1994. December 1994. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/21416.html.

———. *Information technology -- Small Computer System Interface (SCSI) -- Part 222: Fibre Channel Protocol for SCSI, Second Version (FCP-2)*. ISO/IEC 14776-222:2005. February 2005. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/38379.html.

———. *Information technology -- Small Computer System Interface (SCSI) -- Part 232: Serial Bus Protocol-2 (SBP-2)*. ISO/IEC 14776:232:2001. November 2001. Retrieved 13 August 2019. Available for purchase at https://www.iso.org/standard/32329.html.

International Telecommunication Union Telecommunication Standardization Sector. "Information technology - Generic coding of moving pictures and associated audio information: Video." ITU-T H.262. February 2012. May be superseded by update. Retrieved 13 August 2019. Available for purchase at http://www.itu.int/rec/T-REC-H.262-201202-I/en.

Internet Engineering Task Force. "Network Time Protocol (Version 3) Specification, Implementation and Analysis." RFC 1305. March 1992. Obsoleted by RFC 5905. Retrieved 13 August 2019. Available at https://datatracker.ietf.org/doc/rfc1305/.

Microsoft. *Plug and Play Design Specification for IEEE 1394*. Version 1.0b. October 1997. May be superseded by update. Retrieved 13 August 2019. Available at http://www.osdever.net/documents/PNP-Firewire-v1.0b.pdf.

Motion Imagery Standards Board. *Motion Imagery Standards Profile*. MISP-2016-2. February 2016. May be superseded by update. Retrieved 9 August 2016. Available at http://www.gwg.nga.mil/misb/docs/misp/MISP-2016.2.pdf.

North Atlantic Treaty Organization. *NATO Advanced Data Storage Interface Requirements and Implementation Guide*. AEDB-6 Ed. B V. 1. December 2014. May be superseded by update. Retrieved 13 August 2019. Available at https://nso.nato.int/nso/zPublic/ap/PROM/AEDP-06%20EDB%20V3%20E.pdf.

———. *NATO Secondary Imagery Format (NSIF)*. AEDP-04 Ed. 2. V. 1. May 2013. May be superseded by update. Retrieved 13 August 2019. Available at https://nso.nato.int/nso/zPublic/ap/aedp-4(2).pdf.

Range Commanders Council. *Test Methods For Telemetry Systems And Subsystems Volume 1 – Test Methods for Vehicle Telemetry Systems*. 118-12 V. 1. May 2012. May be superseded by update. Retrieved 13 August 2019. Available at https://www.trmc.osd.mil/wiki/download/attachments/91391275/118-20_V1_Vehicle_TM_Systems.pdf.

*** * * END OF STANDARD * * ***