TELEMETRY GROUP

DOCUMENT 123-09

IRIG 106-07 CHAPTER 10 PROGRAMMING HANDBOOK

ABERDEEN TEST CENTER
DUGWAY PROVING GROUND
ELECTRONIC PROVING GROUND
HIGH ENERGY LASER SYSTEMS TEST FACILITY
KWAJALEIN/REAGAN TEST SITE
NATIONAL TRAINING CENTER
WHITE SANDS MISSILE RANGE
YUMA PROVING GROUND

NAVAL AIR WARFARE CENTER AIRCRAFT DIVISION
NAVAL AIR WARFARE CENTER WEAPONS DIVISION
NAVAL UNDERSEA WARFARE CENTER DIVISION, KEYPORT
NAVAL UNDERSEA WARFARE CENTER DIVISION NEWPORT
PACIFIC MISSILE RANGE FACILITY

30TH SPACE WING
45TH SPACE WING
AIR ARMAMENT CENTER
AIR FORCE FLIGHT TEST CENTER
ARNOLD ENGINEERING DEVELOPMENT CENTER
BARRY M. GOLDWATER RANGE

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

**DISTRIBUTION A: APPROVED FOR PUBLIC RELEASE.
DISTRIBUTION IS UNLIMITED.**

This page intentionally left blank.

DOCUMENT 123-09

IRIG 106 CHAPTER 10 PROGRAMMING HANDBOOK

MARCH 2009

Prepared by

TELEMETRY GROUP

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

LIST OF FIGURES v

LIST OF TABLES vi

PREFACE vii

ACRONYMS ix

CHAPTER 1: SCOPE 1-1
1.1 General 1-1
1.2 Document Layout 1-1
1.3 Document conventions 1-2

CHAPTER 2: APPLICABLE DOCUMENTS 2-1
2.1 Government documents 2-1
2.2 Non-government publications 2-1

CHAPTER 3: RECORDER SETUP AND CONFIGURATION 3-1

CHAPTER 4: DATA RETRIEVAL 4-1
4.1 Small Computer Systems Interface (SCSI) Protocol 4-1
4.2 Software Interface 4-6
4.3 Standardization Agreement (STANAG) 4575 Directory 4-9

CHAPTER 5: RECORDER COMMAND AND CONTROL 5-1
5.1 Serial Control 5-1
5.2 Network Control 5-1

CHAPTER 6: DATA FILE INTERPRETATION 6-1
6.1 Overall data file organization 6-1
6.2 Overall data packet organization 6-2
6.3 Required header 6-4
6.4 Optional secondary header 6-5
6.5 Data payload 6-6
6.6 Time Interpretation 6-36
6.7 Index and Event Records 6-37
6.8 Data Streaming 6-37

CHAPTER 7: CONFORMANCE TO IRIG 106 7-1

APPENDIX A: SELECTED SOURCE CODE FILES A-1

APPENDIX A-1 - IRIG106CH10.H..A-1-1
APPENDIX A-2 - IRIG106CH10.C..A-2-1
APPENDIX A-3 - I106_TIME.H A-3-1
APPENDIX A-4 - I106_TIME.C A-4-1
APPENDIX A-5 - I106_DECODE_TIME.H A-5-1
APPENDIX A-6 - I106_DECODE_TIME.C A-6-1
APPENDIX A-7 - I106_DECODE_TMATS.H A-7-1
APPENDIX A-8 - I106_DECODE_TMATS.C A-8-1
APPENDIX A-9 - CONFIG.H A-9-1
APPENDIX A-10 - STDINT.H A-10-1

APPENDIX B: EXAMPLE PROGRAM - CALCULATE HISTOGRAM..................... B-1

APPENDIX C: EXAMPLE PROGRAM - DECODE TMATS... C-1

LIST OF FIGURES

Figure 3-1. Example TMATS attribute tree. 3-2
Figure 3-2. TMATS attribute parser example code. 3-3
Figure 4-1. SCSI INQUIRY CDB structure. 4-2
Figure 4-2. SCSI CDB Control field structure. 4-2
Figure 4-3. SCSI INQUIRY data structure. 4-3
Figure 4-4. SCSI READ CAPACITY CDB structure. 4-4
Figure 4-5. SCSI READ CAPACITY data structure. 4-4
Figure 4-6. SCSI READ(10) CDB structure. 4-5
Figure 4-7. SCSI_PASS_THROUGH structure. 4-8
Figure 4-8. STANAG 4575 Directory Block structure. 4-10
Figure 4-9. STANAG 4575 File Entry structure 4-10
Figure 4-10. STANAG 4575 directory reading and decoding algorithm. 4-11
Figure 6-1. Data packet organization. 6-3
Figure 6-2. Packet header structure. 6-4
Figure 6-3. Optional secondary header structure with IRIG 106 Ch 4 time representation. 6-5
Figure 6-4. Optional secondary header structure with IEEE 1588 time representation. 6-6
Figure 6-5. Intra-packet Time Stamp, 48 bit RTC. 6-6
Figure 6-6. Intra-packet Time Stamp, IRIG 106 Ch 4 binary. 6-6
Figure 6-7. Intra-packet Time Stamp, IEEE 1588. 6-7
Figure 6-8. Type 0x00 Computer Generated Data Format 0 (User) CSDW. 6-7
Figure 6-9. Type 0x01 Computer Generated Data Format 1 (Setup) CSDW 6-7
Figure 6-10. Type 0x02 Computer Generated Data Format 2 (Events) CSDW. 6-8
Figure 6-11. Type 0x02 Computer Generated Data Format 2 (Events) message without optional data. 6-8
Figure 6-12. Type 0x02 Computer Generated Data Format 2 (Events) message with optional data. 6-9
Figure 6-13. Type 0x02 Computer Generated Data Format 2 (Events) message data. 6-9
Figure 6-14. Type 0x03 Computer Generated Data Format 3 (Index) CSDW 6-10
Figure 6-15. Type 0x03 Computer Generated Data Format 3 (Index) Node Index Entry. 6-11
Figure 6-16. Type 0x09 PCM Data Format 1 CSDW. 6-12
Figure 6-17. Type 0x09 PCM Data Format 1 intra-packet data header. 6-13
Figure 6-18. Type 0x11 Time Data Format 1 CSDW. 6-13
Figure 6-19. Type 0x11 Time Data Format 1 structure, day format. 6-14
Figure 6-20. Type 0x11 Time Data Format 1 structure, DMY format 6-15
Figure 6-21. Type 0x19 MIL-STD-1553 Data Format 1 CSDW. 6-15
Figure 6-22. Type 0x19 MIL-STD-1553 Data Format 1 intra-packet header. 6-16
Figure 6-23. 1553 Message word layout. 6-17
Figure 6-24. 1553 Broadcast message word layout. 6-17
Figure 6-25. Algorithm to determine 1553 data word count. 6-17
Figure 6-26. 16PP194 Message transaction. 6-18
Figure 6-27. Type 0x1A MIL-STD-1553 Data Format 2 (16PP194) CSDW. 6-18
Figure 6-28. 16PP194 to IRIG 106 Ch 10 data bit mapping. 6-19
Figure 6-29. 16PP194 Word layout. 6-19
Figure 6-30. 16PP194 Transaction layout. 6-20

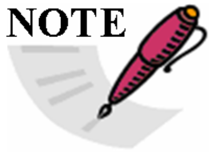
Figure.6-31. Type 0x21 Analog Data Format 1 CSDW. 6-20
Figure 6-32. Type 0x29 Discrete Data Format 1 CSDW. 6-21
Figure 6-33. Type 0x29 Discrete Data Format 1 message. 6-21
Figure 6-34. Type 0x30 Message Data Format 0 CSDW. 6-22
Figure 6-35. Type 0x30 Message Data Format 0 intra-packet header. 6-23
Figure 6-36. Type 0x60 ARINC 429 Data Format 0 CSDW. 6-23
Figure 6-37. Type 0x38 ARINC 429 Data Format 0 intra-packet data header. 6-24
Figure 6-38. Type 0x38 ARINC 429 data format. 6-24
Figure 6-39. Type 0x40 Video Data Format 0 CSDW. 6-25
Figure 6-40. Type 0x40 Video Data Format 1 CSDW. 6-26
Figure 6-41. Type 0x40 Video Data Format 2 CSDW. 6-27
Figure 6-42. Type 0x48 Image Data Format 0 CSDW. 6-28
Figure 6-43. Type 0x49 Image Data Format 1 CSDW. 6-29
Figure 6-44. Type 0x49 Image Data Format 1 intra-packet header. 6-29
Figure 6-45. Type 0x50 UART Data Format 0 CSDW. 6-30
Figure 6-46. Type 0x40 UART Data Format 0 intra-packet data header. 6-30
Figure 6-47. Type 0x58 IEEE 1394 Data Format 0 CSDW. 6-31
Figure 6-48. Type 0x58 IEEE 1394 Data Format 1 CSDW. 6-32
Figure 6-49. Type 0x59 IEEE 1394 Data Format 1 intra-packet data header. 6-33
Figure 6-50. DCRsi interface. 6-34
Figure 6-51. Type 0x60 Parallel Data Format 0 CSDW. 6-34
Figure 6-52. Type 0x68 Ethernet Data Format 0 CSDW. 6-35
Figure 6-53. Type 0x68 Ethernet Data Format 0 intra-packet data header. 6-36
Figure 6-54. UDP Transfer Header, non-segmented data. 6-38
Figure 6-55. UDP Transfer Header, segmented data. 6-39

LIST OF TABLES

Table 1-1. Standard Sized Variable Types 1-2
Table 1-2. Hungarian Notation Prefixes 1-2
Table 7-1. Physical Interface Requirements 7-2
Table 7-2. Logical Interface Requirements 7-3

PREFACE

This programming handbook presents the results of work performed by members of the Telemetry Group (TG) under Task TG-83 of the Range Commanders Council (RCC). This document provides information to assist programmers in developing software for use with RCC Document 106, *Telemetry Standard*, historically referred to as Inter-range Instrumentation Group (IRIG 106). Therefore this document contains frequent references to the IRIG 106 document, primarily Chapter 10, *Digital Recording Standard*; it also covers aspects of Chapter 6 and Chapter 9.

	This handbook is a new document based on (and correlated to) the IRIG 106-07 document.

The RCC gives special acknowledgement for production of this document to:

Task Lead: Mr. Al Berard
846th Test Support Squadron (TSS/TSI)
107 N. Barrancas Avenue, Suite 105
Eglin Air Force Base (AFB), FL 32542
E-Mail: alfredo.berard@eglin.af.mil

Please direct any questions to:

Secretariat, Range Commanders Council
ATTN: TEDT-WS-RCC
100 Headquarters Avenue
White Sands Missile Range, New Mexico 88002-5110
Telephone: (575) 678-1107, DSN 258-1107
E-Mail: wsmrrcc@conus.army.mil

Thid page intentionally left blank.

ACRONYMS

ANSI	American National Standards Institute
API	Application Programming Interface
ARINC	Aeronautical Radio, Incorporated
CCM	Command and Control Mnemonic
CD	Collision Detection
CDB	Command Descriptor Block
CSDW	Channel Specific Data Word
CSMA	Carrier-Sense Multiple Access
CSR	Control and Status Register
DCRsi	Digital Cartridge Recording System (a recording method and digital data interface)
DITS	Digital Information Transfer System
DoD	Department of Defense
ECL	Emitter-coupled Logic
FCP	Fibre Channel Protocol
FCPL	Fibre Channel Private Loop
FC-PLDA	Fibre Channel Private Loop SCSI Direct Attach
GCC	GNU Compiler Collection
GPS	Global Positioning System
HSDB	High Speed Data Bus
I/O	Input/Output
IC	Intelligence Community
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineers
IETF	Internet Engineering Task Force
IOCTL	I/O control
IP	Internet Protocol
IRIG	Inter-range Instrumentation Group
iSCSI	Internet Small Computer Systems Interface
iSNS	Internet Storage Name Service
ISO	International Organization for Standards
IT	Information Technology
ITU	International Telecommunications Union
IU	Information Unit
KLV	Key-Length-Value
LBA	Logical Block Address
LSB	Least Significant Bit
LUN	Logical Unit Number
MAC	Media Access Control
MAS	Military Agency for Standardization
MBR	Master Boot Record
MISB	Motion Imagery Standards Board
MISP	Motion Imagery Standards Profile
MPEG	Moving Picture Experts Group
MUX	Multiplexer

NADSI	NATO Advanced Data Storage Interface
NATO	North Atlantic Treaty Organization
NSG	National System for Geospatial Intelligence
NSIF	NATO Secondary Imagery Format
ORB	Operation Request Block
OS	Operating System
PCM	Pulse Code Modulation
PDU	Protocol Data Unit
PS	Program Stream
RCC	Range Commanders Council
RFC	Request For Comment
RIU	Remote Interface Unit
RMM	Removable Memory Module
RS	Recommended Standard
RTC	Relative Time Counter
SAM	SCSI Architecture Model
SBC	SCSI Block Command
SBP	Serial Bus Protocol
SCSI	Small Computer Systems Interface
SD	Standard Definition
SLP	Service Location Protocol
SPC	SCSI Primary Commands
SPT	SCSI Pass Through
SRB	SCSI Request Block
STANAG	Standardization Agreement
TCP	Transmission Control Protocol
TM	Telemetry
TMATS	Telemetry Attributes Transfer Standard
TS	Transport Stream
UART	Universal Asynchronous Receiver and Transmitter
UDP	User Datagram Protocol
WMUX	Weapons MUX
XML	Extensible Markup Language

CHAPTER 1

SCOPE

1.1 General

The Telemetry Group (TG) of the Range Commanders Council (RCC) developed the *Inter-range Instrumentation Group (IRIG) 106* standard for test range telemetry (TM). The primary purpose of the IRIG 106, published as RCC Document 106, *Telemetry Standard*, is to define a common framework for test range instrumentation 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 ten 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. Chapter 10 defines the interfaces and operational requirements for digital data recording devices. Chapter 10 also references elements of Chapter 6 (Digital Cassette Helical Scan Recorder/Reproducer, Multiplexer/Demultiplexer, Tape Cassette, and Recorder Control and Command Mnemonics Standards) and Chapter 9 (Telemetry Attributes Transfer Standard).

Chapter 10 is comprehensive in its scope. The purpose of this programming handbook is to serve as an adjunct to the IRIG 106 standard to assist the computer programmer when writing software for operating IRIG 106 Chapter 10 standard digital recorders, and when analyzing data from these recorders. A prior working knowledge of Chapter 9, Chapter 10, and applicable sections of Chapter 6, is essential.

1.2 Document Layout

This programming handbook addresses specific topics of Chapter 6, Chapter 9, and Chapter 10 of IRIG 106 which are 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 as follows:

a. Chapter 3: Reading and interpreting Chapter 9 recorder configuration files.
b. Chapter 4: Data retrieval over the standard recorder interfaces.
c. Chapter 5: Recorder command and control.
d. Chapter 6: Data file interpretation.
e. Chapter 7: IRIG 106 standard conformance issues.
f. Appendix A: Selected Source Code Files
g. Appendix B: Example Program - Calculate Histogram
h. Appendix C: Example Program - Decode Telemetry Attributes Transfer Standard (TMATS)

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. This document, with the source code that accompanies it, defines 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
`sz`	Null terminated array of char
`en`	Enumerated type
`m_`	Variable with module scope
`g_`	Variable with global scope
`su`	Structure variable
`Su`	Structure name
`a`	Array of...
`p`	Pointer to...

CHAPTER 2

APPLICABLE DOCUMENTS

2.1 Government documents

a. Range Commanders Council (RCC) Document 106-07, *Telemetry Standard*.
b. RCC Document 118-06, *Test Methods for Telemetry Systems and Subsystems*.
c. MIL-STD-1553B, *Aircraft Internal Time Division Command/Response Multiplex Data Bus*, United States (U.S.) Department of Defense (DoD), 21 September 1978.
d. MIL-STD-2500B, *National Imagery Transmission Format Version 2.1 for the National Imagery Transmission Format Standard*, U.S. DoD, 22 August 1997.
e. 16PP362A, *Weapons Multiplexer (WMUX) Protocol(revision C)*, 46 Test Wing (46TW), Eglin Air Force Base (AFB).
f. Standardization Agreement (STANAG) No. 4545, *North Atlantic Treaty Organization (NATO) Secondary Imagery Format (NSIF)*, Military Agency for Standardization (MAS) / NATO, 27 November 1998.
g. STANAG No. 4575, *NATO Advanced Data Storage Interface (NADSI)*, MAS / NATO.
h. *Motion Imagery Standards Profile (MISP, Version 4.4)*, DoD / Intelligence Community / National System for Geospatial Intelligence (DoD/IC/NSGI), Motion Imagery Standards Board (MISB), 13 December 2007.

2.2 Non-government publications

a. Institute of Electrical and Electronic Engineers (IEEE) Standard 802.3 - 2005, *Carrier-Sense Multiple Access/Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*.
b. IEEE Standard 1394b - 2002, *IEEE Standard for a High-Performance Serial Bus - Amendment 2*.
c. International Organization for Standards /International Electrotechnical Commission (ISO/IEC) 13818-1:1996, *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Systems.*
d. ISO/IEC 13818-2:1996, *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Video*.
e. ISO/IEC 13818-3:1996, *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Audio*.
f. ISO/IEC 13213:1994, *Control and Status Register (CSR) Architecture for Microcomputer Buses.*
g. ISO/IEC 14776-412:2006, *Small Computer System Interface (SCSI) Architecture Model 2 (SAM-2).*
h. ISO/IEC 14776-452:2005, *SCSI Primary Commands 2 (SPC-2).*
i. ISO/IEC 14776-322:2007, *SCSI Block Commands 2 (SBC-2).*
j. ISO/IEC 14776-232:2001, *Serial Bus Protocol 2 (SBP-2).*
k. ISO/IEC 14776-222:2005, *Fibre Channel Protocol for SCSI, Second Version (FCP-2).*

l. International Telecommunications Union, Telecommunication Standardization Sector (ITU-T) Rec. H.262, *Information Technology - Generic Coding of Moving Pictures and Associated Audio Information: Video, AMENDMENT 1: Content Description Data*, 2000.
m. ITU-T Rec. H.264, *Advanced Video Coding for Generic Audiovisual Services*, 2007.
n. *Plug and Play Design Specification for IEEE 1394 Ver. 1.0b*, Microsoft Corporation, 17 October, 1997.
o. Aeronautical Radio, Incorporated (ARINC) 429P1-17, *Mark 33 Digital Information Transfer System (DITS)*, 2004.

CHAPTER 3

RECORDER SETUP AND CONFIGURATION

Chapter 9 of IRIG 106 defines the Telemetry Attributes Transfer Standard (TMATS). Historically, TMATS 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 text is designed to be user friendly and it is also 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. Therefore, an attribute has the form:

```
CODE:DATA;
```

Note: Although not required, lines may be terminated with a carriage return and line feed to improve readability.

TMATS attributes are logically arranged in a hierarchical tree structure. Figure 9-1 in the IRIG Chapter 9 standard shows this attribute tree structure. Unlike other markup languages, such as Extensible Markup Language (XML), 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`" attribute such as "`R-1\ID:MyDataSource;`" with the corresponding "`G\DSI-n`" attribute such as "`G\DSI-1:MyDataSource;`". An example of a portion of a TMATS attribute tree is shown in Figure 3-1. Chapter 9 defines the specific linking fields for the various levels of the TMATS attribute tree.

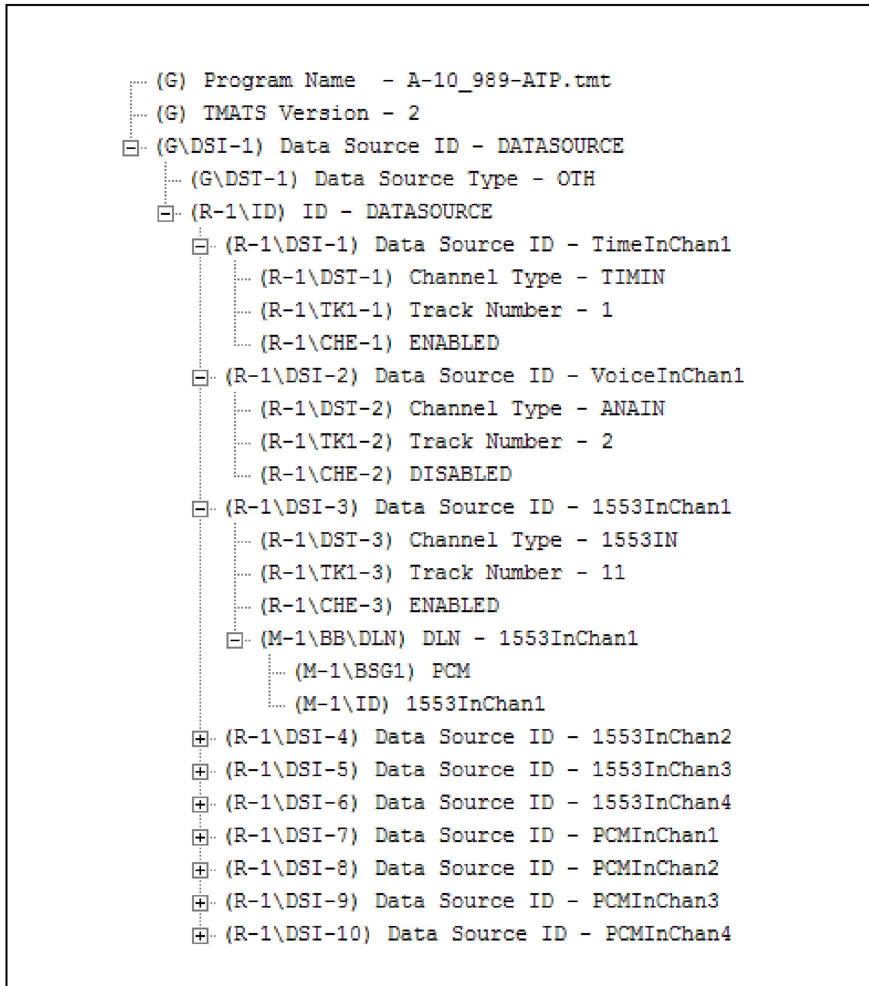


Figure 3-1. Example TMATS attribute tree.

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 3-2. 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 C.

```
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 3-2. TMATS attribute parser example code.

The two basic types of attribute code names are 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 of IRIG 106 identifies quite a few required Recorder 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 the "G\106" attribute. This attribute identifies the version of IRIG 106 to use when interpreting the TMATS information. This attribute only identifies the TMATS version. The Chapter 10 standard version is specified in the TMATS setup record in the recorded data file described in Paragraph 6.5.2 of this document.

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 requirement 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 4

DATA RETRIEVAL

4.1 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 standard 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 Small Computer Systems Interface (SCSI) block transfer protocol. Overall SCSI operation is defined in the SCSI *Architecture Model 2 (SAM-2)* document. Common commands are defined in the SCSI *Primary Commands (SPC-2)* document. Commands for block devices are defined in SCSI *Block Commands 2 (SBC-2)* 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). Logical Unit Numbers 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 SAM defines a Command Descriptor Block (CDB) structure along with associated user input and output buffers. The CDBs are used to issue commands to SCSI devices. 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. A CDB can be 6, 10, 12, or 16 bytes in length. Chapter 10 currently only requires 6 and 10-byte CDBs. Note that multi-byte CDB values such as a 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 4-1. The structure for the Control field is shown in Figure 4-2. The data download interface is required to support the standard INQUIRY response shown in Figure 4-3. 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 4-1. 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 4-2. 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;    // Asynchronous event reporting
cap
    uint8_t       uAddLength;            // Length of additional
parameters
    uint8_t       uReserved3    :  7;
    uint8_t       bSCCS         :  1;    // Embedded storage array
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 4-3. 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 4-4. This command returns the number of available logical blocks and the logical block size in bytes, shown in Figure 4-5. 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 4-4. SCSI READ CAPACITY CDB structure.

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

Figure 4-5. 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 4-6. 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 4-6. SCSI READ(10) CDB structure.

4.1.1 IEEE 1394. IEEE 1394 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 (ORB). The ORBs encapsulate the information in SCSI CDBs and input/output buffers, as well as additional information needed by the 1394 bus. The 1394 bus operates by exposing a shared memory address space, negotiated during device initialization. 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. LUN 1 is used for interfacing to the RMM Real Time Clock.

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

4.1.2 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). FCP defines several types of IUs used to communicate with SCSI targets. Fibre Channel also defines the Fibre Channel Private Loop (FCPL) SCSI Direct Attach (FC-PLDA) protocol. The FC-PLDA further defines the implementation of SCSI over Fibre Channel. Chapter 10 requires conformance to FC-PLDA.

4.1.3 Internet Small Computer Systems Interface (iSCSI). The Internet Engineering Task Force (IETF) has published Request For Comment (RFC) 3270 defining the iSCSI protocol for transporting CDBs over Transmission Control Protocol/Internet Protocol (TCP/IP) based network to an 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). The PDUs encapsulate the information in SCSI CDBs and input/output 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, the TCP port number, and the iSCSI target name information. There are several methods that may be used to find targets. An iSCSI supports the following discovery mechanisms:

a. 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 information to establish a TCP connection; it also uses the iSCSI target name information to establish an iSCSI session. This discovery option is convenient for small iSCSI setups.
b. 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).
c. 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 (SLP [RFC2608] and Internet Storage Name Service (iSNS [iSNS]) 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. For command and control, LUN 1 or LUN 33 is used.

4.2 Software Interface

All recorder data download interfaces appear as SCSI block Input/Output (I/O) devices, and respond to the subset of SCSI commands set forth in Chapter 10. However, different operating systems provide vastly different types of Application Programming Interfaces (API) 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, to initialize and configure them, and then offer a single programming interface for user application programs.

The interface used by Windows application 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 (IOCTL) code.

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. Note that Windows may require security priviledges above those normally granted to a regular user to call `CreateFile()`. 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 the SCSI port name is used, the Win32 application must set the proper `PathId`, `TargetId`, and `LUN` in the SCSI pass through structure.

Once a valid handle to a SCSI device is obtained, then appropriate input and output 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; these include:

a. `IOCTL_SCSI_GET_INQUIRY_DATA`
b. `IOCTL_SCSI_GET_CAPABILITIES`
c. `IOCTL_SCSI_PASS_THROUGH`
d. `IOCTL_SCSI_PASS_THROUGH_DIRECT`

The `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`.

The `SCSI_INQUIRY_DATA` structure contains a member named DeviceClaimed. DeviceClaimed indicates whether or not a class driver has claimed this particular SCSI device. If a device is claimed, all SCSI pass through 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 SCSI pass through requests are sent directly to the SCSI port driver.

For `IOCTL_SCSI_GET_INQUIRY_DATA`, no data is sent to the device; data is only read from the device. 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.

The 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 (SRB), 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 Command Descriptor Blocks (CDB) 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.

The IOCTL_SCSI_PASS_THROUGH uses the SCSI_PASS_THROUGH structure. The IOCTL_SCSI_PASS_THROUGH_DIRECT uses the SCSI_PASS_THROUGH_DIRECT structure.

The SCSI_PASS_THROUGH structure is shown in Figure 4-7.

```
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 4-7. 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 SCSI pass through 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 input and output 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 logical unit number 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 sixteen remaining bytes are for the CDB data. The format of this data must conform to the SCSI-2 standard.

4.3 Standardization Agreement (STANAG) 4575 Directory

Chapter 10 recorders and RMMs 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 NATO Advanced Data Storage Interface (NADSI) “sees” a single directory of files in contiguous logical addresses.

Disk blocks are accessed by Logical Block Address (LBA). It is common in many operating systems (OSs) and disk structures for block 0 to be a Master Boot Record (MBR). An MBR typically contains OS 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 logical block 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 4-8. IRIG 106-03 and IRIG 106-04 defined the STANAG 4575 directory data as "little-endian." Subsequent IRIG 106 versions have defined directory data as "big-endian." Applications can test the **achMagicNumber** and **uRevLink** values to determine whether big-endian or little-endian is being used. If big-endian is being used, 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 4-9. 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 4-8. 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 4-9. 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** equals the current LBA. This algorithm is shown in Figure 4-10.

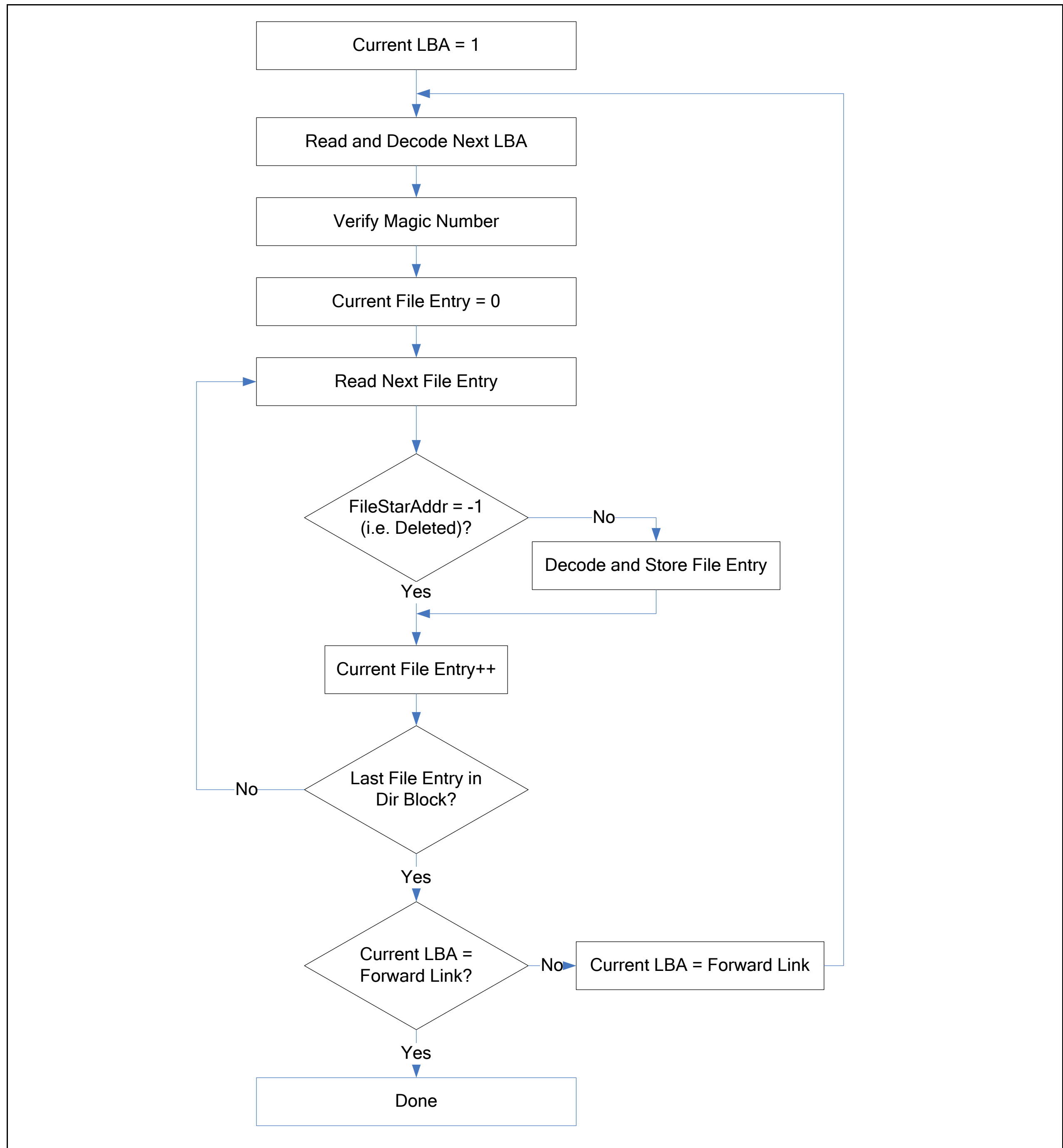


Figure 4-10. STANAG 4575 directory reading and decoding algorithm.

This page intentionally left blank.

CHAPTER 5

RECORDER COMMAND AND CONTROL

Recorders are controlled over a full duplex communications channel. Typically, this channel is either a Recommended Standard 232 (RS-232) or an RS-422 serial communications port. Chapter 10 of IRIG 106 also defines a mechanism which allows control over any of the supported recorder network channels, including Ethernet, Fibre Channel, and IEEE 1394b.

The recorder Command and Control Mnemonic (CCM) 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 are complete is to issue multiple `.STATUS` commands, checking the status complete field until it indicates the command has completed processing.

5.1 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. Commands are not case sensitive.

Neither Chapter 6 nor Chapter 10 addresses serial flow control. Most commands generate very little text, and buffer overflow shouldn't be a problem. However, the `.TMATS` command can result in a considerable amount of text being 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.

5.2 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 command are described in the SCSI Primary Commands document in paragraph 2.2h.

This page intentionally left blank.

CHAPTER 6

DATA FILE INTERPRETATION

6.1 Overall Data File Organization

Chapter 10 data files are organized as a sequential series of data packets. Each data packet 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 in a data file be an IRIG 106 Chapter 9 format, Telemetry Attributes Transfer Standard (TMATS) packet. The TMATS packet is used to configure the recorder and to describe the data being recorded. Also, TMATS is stored in a Computer Generated Data, Format 1 (Data Type = 0x01) data packet and is discussed in paragraph 6.5.2.

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. This value should not be confused with the Data Type Version (but sometimes called Header Version) discussed in paragraph 6.3. The IRIG 106 Version field is only defined for the IRIG 106-07 and later versions of the IRIG 106 standard. Since unused reserved fields must be initialized to zero, this field will have a value of 0 in data files compliant with IRIG 106 prior to IRIG 106-07.

Starting with publication of IRIG 106-07, more than one TMATS packet is allowed at the beginning of the file if recorder configuration information exceeds the packet size limit.

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 that a dynamic data packet means 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 to verify a valid time packet has been received before attempting to interpret the RTC as described in paragraph 6.6.

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) with the maximum size of 524,288 bytes. Computer Generated Data, Format 1 (TMATS) packets have a maximum packet size of 134,217,728 bytes. To provide this 4 byte alignment, padding bytes are added, if necessary, to the end of a data packet, just before the checksum. 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 a 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 in which they are received, but data packets and data messages can occur in the data file out of time order. This order 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. Therefore, 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 received 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 A. An example program that demonstrates reading and interpreting a Chapter 10 file can be found in Appendix B.

6.2 Overall Data Packet Organization

Overall data packet organization is shown in Figure 6-1. 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 6-1. Data packet organization.

Data packets must contain data. They are not allowed to only contain filler. 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 the same length of packets from a particular channel. The standard does not expressly prohibit filler after the packet trailer but before the next data packet header; however, inserting filler after the last trailer is considered bad practice. Still, when reading data packets, the programmer should 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 a position 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:

a. 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 which is an integer multiple of four. However, if the data file is corrupted then the sync pattern may not fall on the normal four byte boundary. Scan the file a byte at a time, ignoring the normal four byte alignment. When the Sync pattern is found then
b. Calculate and test the header checksum.
c. If a secondary header exists, calculate and test the secondary header checksum.
d. Calculate and test the data checksum.

If 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.

6.3 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 6-2.

```
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       ubyDataVer;            // Data Version
    uint8_t       ubySeqNum;             // Sequence Number
    uint8_t       ubyPacketFlags;        // Packet Flags
    uint8_t       ubyDataType;           // Data type
    uint8_t       aubyRelTime[6];        // Reference time
    uint16_t      uChecksum;             // Header Checksum
    };
```

Figure 6-2. 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, "`R-m\TK1.`" The Channel ID field is a 16-bit field. However, the Chapter 9 TMATS format restricts the value of Channel ID to a two digit number (i.e. from 0 to 99). It is anticipated that this TMATS restriction will be lifted in future IRIG 106 standard releases.

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. It is typical for Channel ID 0 to contain Computer Generated Data Format 1 Setup Records (0x01), Computer Generated

Data Format 2 Recording Events Records (0x02), and Computer Generated Data Format 3 Recording Index Records (0x03).

The data payload format is interpreted based on the value of the Data Type field and the Data Version field in the packet header. This field is sometimes incorrectly called "Header Version." Each packet data payload can only contain one type of data (e.g. 1553, PCM, etc.). 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:

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

```
Offset = Packet Length
         - Primary Header Length (24)
         - Secondary Header Length (12) (if included)
```

c. Move read pointer

6.4 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 6-3 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 6-4 is used when secondary header time is to be interpreted as an IEEE 1588 format value (Flag Bits 3-2 = 1).

```
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 6-3. 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 6-4. Optional secondary header structure with IEEE 1588 time representation.

6.5 Data Payload

After the standard header and optional secondary header, each data packet begins with CSDW(s). The length of the CSDW varies depending on the data type. For example, Analog Data Format 1 may have multiple CSDWs. The CSDW 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 are used between messages.

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

Intra-packet headers, when they are present, 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 6-5, Figure 6-6, and Figure 6-7. These three 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 6-5. 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 6-6. Intra-packet Time Stamp, IRIG 106 Ch 4 binary.

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

Figure 6-7. Intra-packet Time Stamp, IEEE 1588.

6.5.1 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 6-8. 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 6-8. Type 0x00 Computer Generated Data Format 0 (User) CSDW.

6.5.2 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 6-9.

```
struct SuTmats_ChanSpec
    {
    uint32_t    iCh10Ver         :  8;      // Recorder Ch 10 Version
    uint32_t    bConfigChange    :  1;      // Recorder config changed
    uint32_t    iReserved        : 23;      // Reserved
    };
```

Figure 6-9. Type 0x01 Computer Generated Data Format 1 (Setup) CSDW.

Note that this structure definition for the CSDW first appeared in IRIG 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 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.

During analysis of previously recorded data, it is useful to be able to quickly locate TMATS setup records, especially setup records that are different from previous setup records in

the data file. The preferred method to locate these setup records is by indexing Computer Generated Data Format 1 packets using the indexing method described in paragraph 6.7.

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

6.5.3 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 6-10. 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 in the intra-packet header.

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

Figure 6-10. Type 0x02 Computer Generated Data Format 2 (Events) CSDW.

There are a number of permutations of the recorded event. 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 6-11. Event data with optional data (**bIntraPckHdr = 1**) is shown in Figure 6-12. The format for the event message itself is shown in Figure 6-13.

```
struct SuEvents
    {
    uint64_t              suIntraPckTime;      // Intra-packet time stamp
    struct SuEvents_Data  suData;              // Data about the event
    };
```

Figure 6-11. Type 0x02 Computer Generated Data Format 2 (Events) message without optional data.

```
struct SuEvents_with_Optional
    {
    uint64_t              suIntraPckTime;       // Intra-packet time stamp
    uint64_t              suIntrPckData;        // Intra-packet data
    struct SuEvents_Data  suData;               // Data about the event
    };
```

Figure 6-12. 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;   //
    };
```

Figure 6-13. Type 0x02 Computer Generated Data Format 2 (Events) message data.

The **suIntraPckTime** field in the data structures of Figure 6-11 and Figure 6-12 (shown above) represents event time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5), or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

If the event message includes the optional **suIntrPckData** Intra-packet Data Header field, shown in the data structure of Figure 6-12, 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 6-18 and Figure 6-19. 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 Intra-packet Data Header. For this reason, this field should be used with caution, if used at all. Data about the recorded event is found in the **SuEvents_Data** structure shown in Figure 6-13. The particular event is identified by the `uNumber` field, which corresponds to 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.

6.5.4 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. Root Index packets are organized as a linked list of packets. Each Root Index packet contains a value for the file offset of the preceding Root Index packet in the linked list. 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"). 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 6-14, 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 intra-packet data header. The **bFileSize** field indicates the presence of the optional file size field. The **uIndexType** field indicates the 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 6-14. 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, an optional intra-packet data header, and a Node Index Entry data structure. The intra-packet time stamp represents indexed packet data time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

If the index message includes the optional Intra-packet Data Header 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 6-18 and Figure 6-19. 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 Intra-packet Data Header. 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 6-15. 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 6-15. 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.

Each Root Index is composed of an intra-packet time stamp, an optional intra-packet data header, and a Node Index data packet offset value. The intra-packet time stamp represents indexed packet data time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5), or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6 or IEEE 1588 time format (format shown in Figure 6-7).

If the Root Index message includes the optional Intra-packet Data Header 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 6-18 and Figure 6-19. 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 Intra-packet Data Header. 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.

6.5.5 Type 0x04 - 0x07, Computer Generated Data, Format 4 - Format 7. Reserved for future use.

6.5.6 Type 0x08, PCM Data, Format 0. Reserved for future use.

6.5.7 Type 0x09, PCM Data, Format 1 (IRIG 106 Chapter 4/8). PCM Data, Format 1 packets are used to record Pulse Code Modulation (PCM) data frames. PCM data is a serial stream of bits from multiple interleaved data sources. Each data source generally operates at a different data rate and have their digital data interleaved in a fixed, defined fashion. The data from these multiplexed data sources are organized into major frames and minor frames. PCM Data

Format 1 records minor frames as integral units of data. In general, the PCM data packet will contain multiple PCM minor frame data messages in packed and unpacked mode. There is extensive discussion of PCM in IRIG 106 Chapter 4, Chapter 8, and Appendix C, as well as RCC Document 119 (Telemetry Applications Handbook).

The PCM minor frame data is recorded in one of three major modes; Unpacked Mode, Packed Mode, and Throughput Mode. 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 16-bit alignment mode, or the next 32-bit boundary in 32-bit boundary mode. In Packed Mode, packing is enabled and padding 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 6-16. 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 Packed 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 intra-packet headers.

```
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 6-16. Type 0x09 PCM Data Format 1 CSDW.

The optional intra-packet header and individual PCM minor frame messages follow the CSDW. The format of the intra-packet data header is shown in Figure 6-17. The **suIntrPckTime** value is an eight byte representation of time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5), or as absolute time. If this time is

absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

The intra-packet header, indicated by **bIntraPckHdr = 1**, is required for unpacked and packed mode. It is prohibited for throughput mode. The **uMajorFrStatus** field indicates the lock status of the major frame. The **uMinorFrStatus** indicates the lock status of the minor frame.

```
struct SuPcmF1_Header
    {
    uint64_t    suIntraPckTime;                // 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 6-17. Type 0x09 PCM Data Format 1 intra-packet data header.

One minor frame of data follows the intra-packet data header. 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 **bAlignment** flag in the CSDW.

6.5.8 Type 0x0A, 0x0F PCM Data, Format 2 - Format 7. Reserved for future use.

6.5.9 Type 0x10, Time Data, Format 0. Reserved for future use.

6.5.10 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 paragraph 6.6.

The time data packet begins with the CSDW shown in Figure 6-18.

```
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    uReserved1  :  2;
    uint32_t    uReserved2  : 16;
    };
```

Figure 6-18. Type 0x11 Time Data Format 1 CSDW.

The **uTimeSrc** field indicates 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 source applied to the recorder. The **uDataFmt** 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 6-19. Time representation in Day-Month-Year format is shown in Figure 6-20. The **bLeapYear** field in the CSDW is useful to convert Day of the Year to Day and Month when the year is not known.

```
// Time message - Day format
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 6-19. Type 0x11 Time Data Format 1 structure, day format.

```
// Time message - DMY 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 6-20. Type 0x11 Time Data Format 1 structure, DMY format

6.5.11 Type 0x12 - 0x17, Time Data, Format 2 - Format 7. Reserved for future use.

6.5.12 Type 0x18, MIL-STD-1553 Data, Format 0. Reserved for future use.

6.5.13 Type 0x19, MIL-STD-1553 Data, Format 1 (MIL-STD-1553B Data). MIL-STD-1553 Data, Format 1 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 6-21. 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 6-21. Type 0x19 MIL-STD-1553 Data Format 1 CSDW.

The individual 1553 messages follow the CSDW. Each 1553 message has an intra-packet time stamp, an intra-packet header data word, and then the actual 1553 message. The layout of the message header is shown in Figure 6-22. The **suIntPktTime** field is an 8-byte value. The specific interpretation of this field is determined by packet header flags. This time is interpreted as a RTC value as depicted in Figure 6-5 if Secondary Headers are not enabled by Packet Flags, Bit 6. If Secondary Headers are enabled, then the format of the intra-packet time stamp is the same as the secondary header, determined by Packet Flags, Bits 2 and 3. These formats are depicted in Figure 6-6 and Figure 6-7. Various bit flags and values are found in the intra-packet data header.

```
// Intra-message header
struct Su1553F1_Header
    {
    uint64_t    suIntPktTime;           // 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 6-22. Type 0x19 MIL-STD-1553 Data Format 1 intra-packet header.

The amount of data that follows the intra-packet header 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 is 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 6-23 and Figure 6-24. 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 6-25. This algorithm only works for a message with no errors. Otherwise, Block Status Word bits and the Length Word are used to determine data and status words actually present.

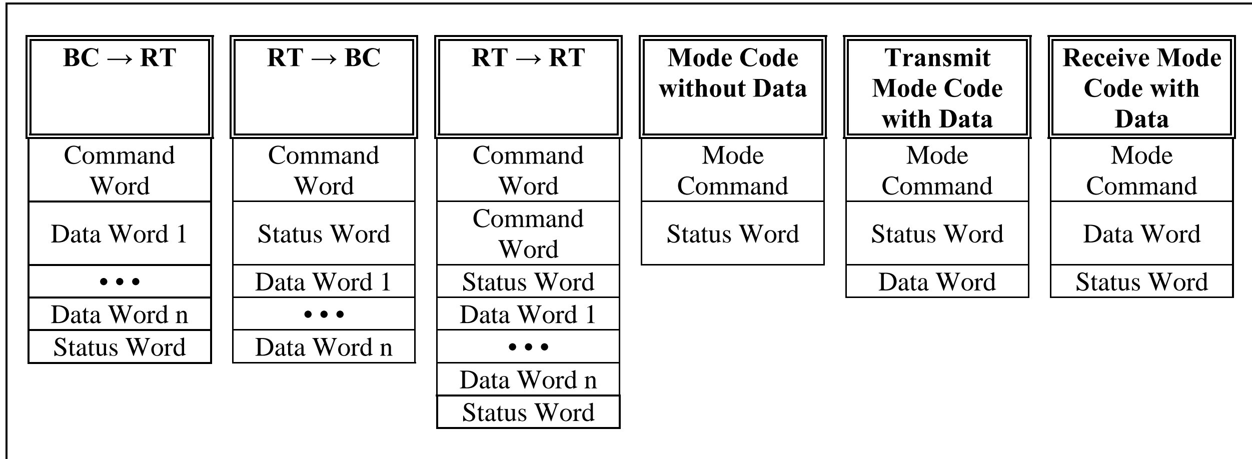


Figure 6-23. 1553 Message word layout.

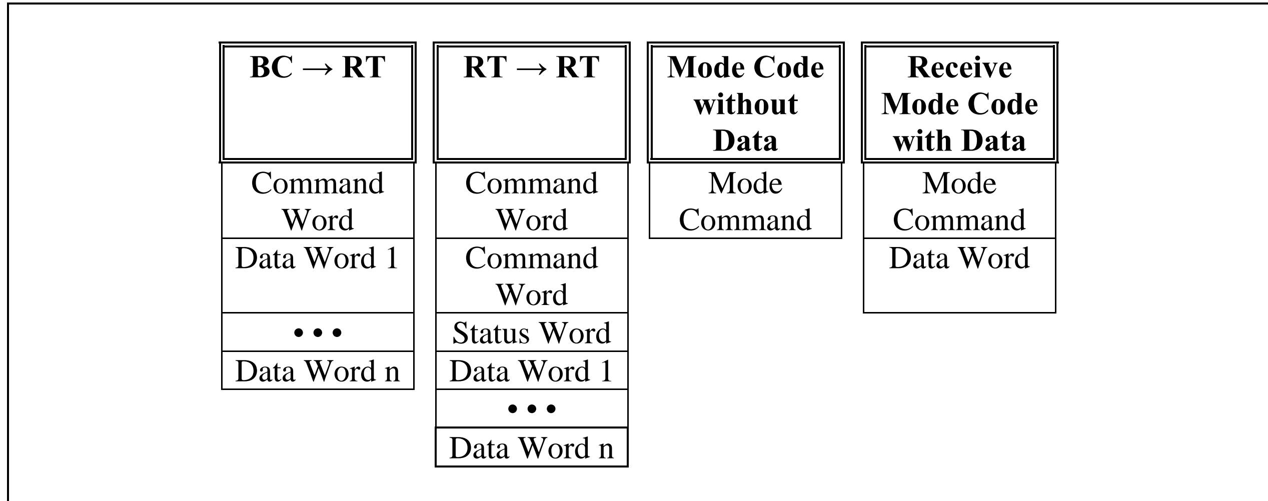


Figure 6-24. 1553 Broadcast message word layout.

```
// 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 6-25. Algorithm to determine 1553 data word count.

6.5.14 Type 0x1A, MIL-STD-1553 Data, Format 2 (16PP194 Bus). MIL-STD-1553, Format 2 packets are 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*. 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 6-26. Multiple transactions may be encoded into the data portion of a single packet.

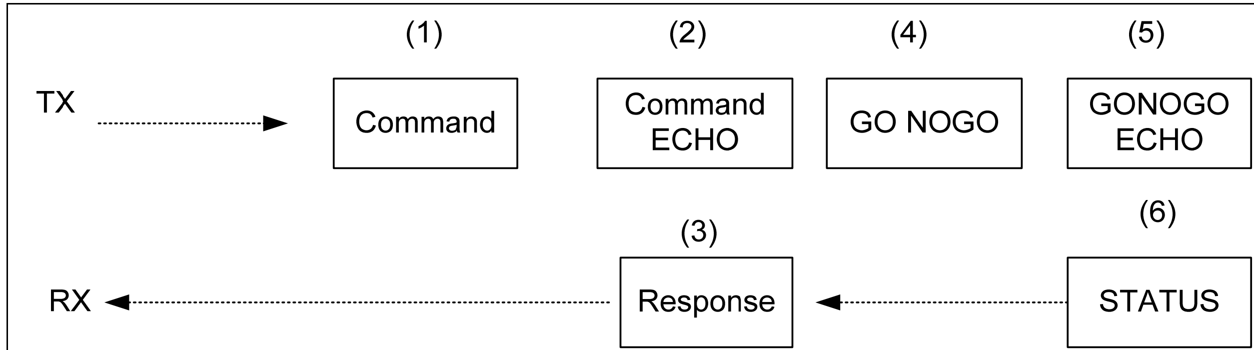


Figure 6-26. 16PP194 Message transaction.

The layout of the CSDW is show in Figure 6-27. 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 6-27. 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, four address bits, four 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 6-28.

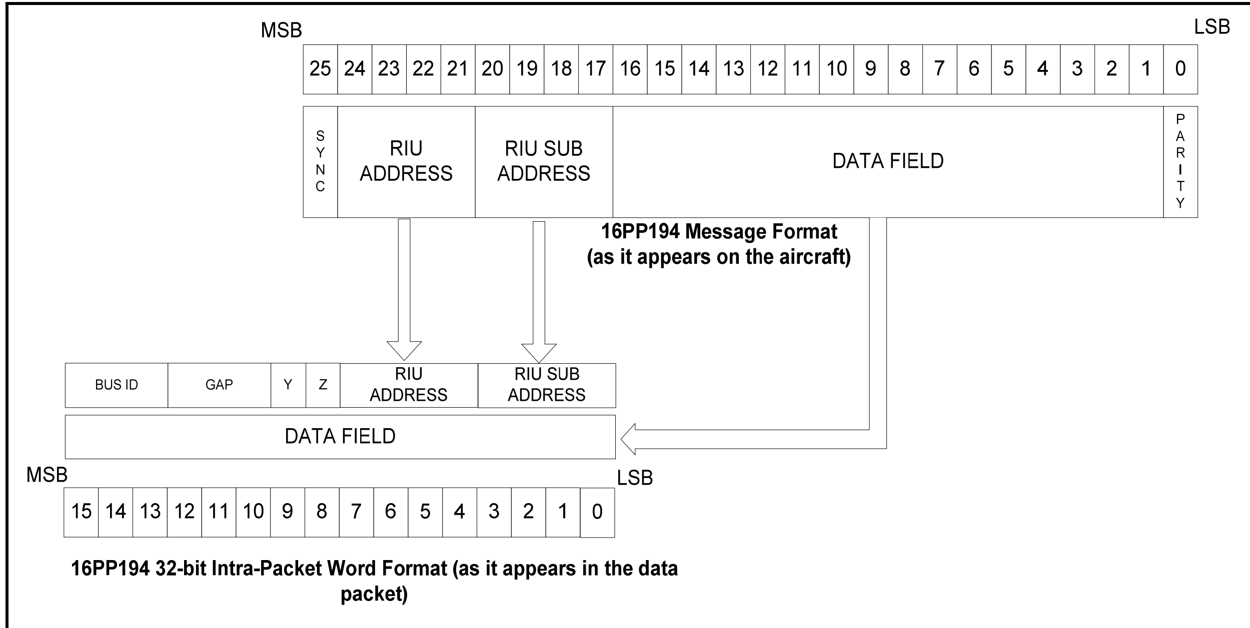


Figure 6-28. 16PP194 to IRIG 106 Chapter 10 data bit mapping.

The layout of the recorded 16PP194 word is shown in Figure 6-29. 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 6-30.

```
struct SuSu16PP194_Word
    {
    uint32_t    uDataWord;      : 16;   // Data word contents
    uint32_t    uRiuSubAddr     : 4;    // Parity error flag
    uint32_t    uRiuAddr        : 4;    // Parity error flag
    uint32_t    bParityError    : 1;    // Parity error flag
    uint32_t    bWordError      : 1;    // Manchester error flag
    uint32_t    uGap            : 3;    // Gap time indicator
    uint32_t    uBusID          : 3;    // Bus ID indicator
    };
```

Figure 6-29. 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 6-30. 16PP194 Transaction layout.

6.5.15 Type 0x1B - 0x1F, MIL-STD-1553 Data, Format 3 - Format 7. Reserved for future use.

6.5.16 Type 0x20, Analog Data, Format 0. Reserved for future use.

6.5.17 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 describes a situation in which each sample occupies one 16-bit word 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 6-31. 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 the 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 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 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.6-31. Type 0x21 Analog Data Format 1 CSDW.

Sample rate, Least Significant Bit (LSB)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.

6.5.18 Type 0x22 - 0x27, Analog Data, Format 2 - Format 7. Reserved for future use.

6.5.19 Type 0x28, Discrete Data, Format 0. Reserved for future use.

6.5.20 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 6-32. 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 Data message 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 6-32. Type 0x29 Discrete Data Format 1 CSDW.

The layout of the Discrete Data message is shown in Figure 6-33. Each message contains a time stamp and the state of the discrete data input signals. The **suIntraPckTime** field in the data structures of represents event time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

```
struct SuDiscreteF1
    {
    uint64_t            suIntraPckTime;        // Intra-packet time stamp
    uint32_t            suData;                // Data about the event
    };
```

Figure 6-33. Type 0x29 Discrete Data Format 1 message.

The IRIG 07 Chapter 10 and prior standard incorrectly states 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. It is anticipated that with will be corrected in a future release

6.5.21 Type 0x2A - 0x2F, Discrete Data, Format 2 - Format 7. Reserved for future use.

6.5.22 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 (HSDB) 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 65,536 (64k) bytes. A Message Data packet can contain multiple data messages, a single data message, or a segment of a large (greater than 64k) 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 6-34. 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;
    } SuMessageF0_ChanSpec;
```

Figure 6-34. Type 0x30 Message Data Format 0 CSDW.

The layout of the Message Data message intra-packet header is shown in Figure 6-35. Each header contains a time stamp and some additional information about the data that follows in the message. The **suIntraPckTime** field in the intra-packet data structure represents event time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7). 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 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. 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 SuMessageF0_Header
    {
    uint64_t    suIntraPckTime;             // 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 6-35. Type 0x30 Message Data Format 0 intra-packet header.

The IRIG 106-07 Chapter 10 standard, and earlier versions, incorrectly states 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. It is anticipated that with will be corrected in a future release

6.5.23 Type 0x31 - 0x37, Message Data, Format 1 - Format 7. Reserved for future use.

6.5.24 Type 0x38, ARINC 429 Data, Format 0 (ARINC 429 Data). The ARINC 429 Data, Format 0 packets are used to record data messages from an ARINC 429 data bus. ARINC 429 is a unidirectional data bus that is 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 6-36. 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 6-36. Type 0x60 ARINC 429 Data Format 0 CSDW.

Individual ARINC 429 data messages follow the CSDW. Each message is preceded with an intra-packet data header followed by the ARINC 429 data word.

The layout of the ARINC 429 data message intra-packet data header is shown in Figure 6-37. 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 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 6-37. Type 0x38 ARINC 429 Data Format 0 intra-packet data header.

The layout of the individual ARINC 429 data work is shown in Figure 6-38. 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 6-38. Type 0x38 ARINC 429 data format.

6.5.25 Type 0x39 - 0x3F, ARINC 429 Data, Format 1 - Format 7. Reserved for future use.

6.5.26 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 Moving Picture Experts Group (MPEG-2) Transport Stream (TS) packets. Video can be encoded with either MPEG-2 Main Profile Main Level (MP@ML) encoding or H.264 (also know as MPEG-4 Part 10 and MPEG-4 AVC) Main Profile Level 3 (MP@L3) encoding. The H.264 encoder is usually the preferred encoder for lower bit rate video. This encoding is usually referred to as Standard Definition (SD), and has a maximum resolution of 720 by 576 pixels but is frequently less.

The layout of the CSDW is shown in Figure 6-39. The **uType** value indicates the specific video encoding type in the MPEG-2 stream. This field was first defined in IRIG 106-07 (Header 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 intra-packet header 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 6-39. Type 0x40 Video Data Format 0 CSDW.

MPEG-2 TS packets follow the CSDW. 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 intra-packet time stamp. The intra-packet time represents TS time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (shown in Figure 6-7). 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 contiguous stream of 1504 bits. 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. TS packet bit 0 (the first TS bit) is the leftmost bit (msb) 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. 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 packet, 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. Chapter 10 requires audio to be encoded with the video stream using audio encoding inherent with MPEG format video streams. There are instances where vendors have encoded audio in a separate analog channel to meet special requirements.

6.5.27 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 13818-1:2000 standard for both Program Streams (PS) and Transport Streams (TS). Any Profile and Level combination can be used in Format 1.

The layout of the CSDW is shown in Figure 6-40. 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 **uEPL** value indicates the video packet encoding Profile and Level used. The **bIntraPckHdr** flag indicates the presence of intra-packet header 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 Key-Length-Value (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 6-40. Type 0x40 Video Data Format 1 CSDW.

MPEG-2 PS or TS packets follow the CSDW. TS packets are a fixed length of 188 bytes, but PS packers are variable length. If the **bIntraPckHdr** flag is set, each MPEG-2 packet will be preceded with an eight byte intra-packet time stamp. The intra-packet time represents MPEG packet time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5), or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

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 a good knowledge of MPEG internal data structures. For this reason, the use of intra-packet headers between MPEG packets should be carefully considered. It could make decoding Format 1 packets quite complicated.

Chapter 10 provides little guidance for the specific layout of the MPEG-2 data payload. In practice, Video Data Format 1 MPEG-2 data is stored in “native” format within the IRIG data packet. The data is byte ordered within the data packet in the same order as the MPEG-2 data packet. Unlike Video Data Format 0 data, Format 1 MPEG-2 packets do not require byte swapping before decoding.

Chapter 10 requires audio to be encoded with the video stream using audio encoding inherent with MPEG format video streams. There are instances where vendors have encoded audio in a separate analog channel to meet special requirements.

6.5.28 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 International Electrotechnical Commission (ISO/IEC) 13818-1:2000 standard for both Program Streams (PS) and Transport Streams (TS), 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 6-41. 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 intra-packet header 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 Key-Length-Value (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 6-41. Type 0x40 Video Data Format 2 CSDW.

MPEG-2 PS or TS packets follow the CSDW. TS packets are a fixed length of 188 bytes, but PS packers are variable length. If the **bIntraPckHdr** flag is set, each MPEG-2 packet will be preceded with an eight byte intra-packet time stamp. The intra-packet time represents MPEG packet time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

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 intra-packet headers between MPEG packets should be carefully considered. It could make decoding Format 2 packets quite complicated.

Chapter 10 provides little guidance for the specific layout of the MPEG-4 data payload. In practice, Video Data Format 2 MPEG-4 data is stored in "native" format within the IRIG data packet. The data is byte ordered within the data packet in the same order as the MPEG-4 data packet. Unlike Video Data Format 0 data, Format 2 MPEG-4 packets do not require byte swapping before decoding.

Chapter 10 requires audio to be encoded with the video stream using audio encoding inherent with MPEG format video streams. There are instances where vendors have encoded audio in a separate analog channel to meet special requirements.

6.5.29 Type 0x43 - 0x47, Video Data, Format 3 - Format 7. Reserved for future use.

6.5.30 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 6-42. The **uLength** value is the number bytes in each segment. The **bIntraPckHdr** flag indicated the presence of the intra-packet data header. 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    bIntraPckHdr    :  1;     // Intra-packet header flag
    uint32_t    uSum            :  2;     //
    uint32_t    uPart           :  2;     //
    };
```

Figure 6-42. Type 0x48 Image Data Format 0 CSDW.

Individual image data messages follow the CSDW. Each message may have an optional intra-packet data header followed by the image data. The intra-packet data header, if present (indicated by **bIntraPckHdr** = 1), is an eight byte representation of time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

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.

Chapter 10 of IRIG 106-07 (and prior editions of the standard) incorrectly state 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. This handbook is based on (and correlates to) the IRIG 106-07 edition; however the "bit 7" error will be corrected in future releases of this handbook.

6.5.31 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 6-43. The **uFormat** value indicates the format of the image data. The **bIntraPckHdr** flag indicated the presence of the intra-packet data header. 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;      //
    uint32_t    uFormat         :  4;      // Image format
    uint32_t    bIntraPckHdr    :  1;      // Intra-packet header flag
    uint32_t    uSum            :  2;      //
    uint32_t    uPart           :  2;      //
    };
```

Figure 6-43. Type 0x49 Image Data Format 1 CSDW.

Individual image data messages follow the CSDW. Each message may have an optional intra-packet data header (indicated by **bIntraPckHdr** = 1) followed by the image data. The format of the intra-packet data header is shown in Figure 6-44. The **suIntrPckTime** value is an eight byte representation of time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7). The uMsgLength value indicates the length of the following still image or segment.

```
struct SuImageF1_Header
    {
    uint64_t    suIntraPckTime;            // Reference time
    uint32_t    uMsgLength;                // Message length
    };
```

Figure 6-44. Type 0x49 Image Data Format 1 intra-packet header.

Chapter 10 of IRIG 106-07 (and prior editions of the standard) incorrectly state 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. This handbook is based on (and correlates to) the IRIG 106-07 edition; however the "bit 7" error will be corrected in future releases of this handbook.

6.5.32 Type 0x4A - 0x4F, Image Data, Format 2 - Format 7. Reserved for future use.

6.5.33 Type 0x50, Universal Asynchronous Receiver and Transmitter (UART) Data, Format 0 (UART Data). UART Data, Format 0 packets are used to record character data from an asynchronous serial interface. Generally, 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 msec 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 6-45. The **bIntraPckHdr** flag is used to indicate if an optional intra-packet time stamp is included with each UART data message.

```
struct SuUartF0_ChanSpec
    {
    uint32_t    uReserved       : 31;
    uint32_t    bIntraPckHdr    :  1
    } SuUartF0_ChanSpec;
```

Figure 6-45. Type 0x50 UART Data Format 0 CSDW.

The UART data message may have an optional intra-packet time stamp. If so, this time field will be an 8 byte value. Time is represented in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

The layout of the UART Data message intra-packet data header is shown in Figure 6-46. 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 intra-packet data header. 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 indicates that errors have occurred in the reception of the data. The recorded data may, therefore, be invalid and unusable.

```
struct SuUartF0_DataHeader
    {
    uint16_t    uDataLength     : 16;    // Num of bytes of UART data
    uint16_t    uSubChannel     : 14;    // Subchannel for following data
    uint16_t    uReserved       : 1;
    uint16_t    bParityError    : 1;     //Parity Error
    };
```

Figure 6-46. Type 0x40 UART Data Format 0 intra-packet data header.

Chapter 10 of IRIG 106-07 (and prior editions of the standard) incorrectly state 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. This handbook is based on (and correlates to) the IRIG 106-07 edition; however the "bit 7" error will be corrected in future releases of this handbook.

6.5.34 Type 0x51 - 0x57, UART Data, Format 1 - Format 7. Reserved for future use.

6.5.35 Type 0x58, IEEE 1394 Data, Format 0 (IEEE 1394 Transaction). IEEE 1394 Data, Format 0 packets are used to record data messages at the transaction layer of the IEEE 1394 serial data bus. Currently IEEE 1394, IEEE 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 PHY packet, a Primary packet, and an Acknowledgement 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 for more general purpose use and are used to record all 1394 packets including PHY, Primary, and Ack packets.

The layout of the CSDW is shown in Figure 6-47. 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 6-47. Type 0x58 IEEE 1394 Data Format 0 CSDW.

Type 0 and Type 1 data packets will not have an intra-packet header. Type 2 data packets will have intra-packet headers between transaction data messages with the data packet. The Type 2 intra-packet header is an 8 byte time value. Time is represented in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5), or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7).

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

Chapter 10 of IRIG 106-07 (and prior editions of the standard) incorrectly state 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. This handbook is based on (and correlates to) the IRIG 106-07 edition; however the "bit 7" error will be corrected in future releases of this handbook.

6.5.36 Type 0x59, IEEE 1394 Data, Format 1 (IEEE 1394 Physical Layer). IEEE 1394 Data, Format 1 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. This Format 1 packet provides more status information, though, and is preferred over the Format 0, Type 2 data packet.

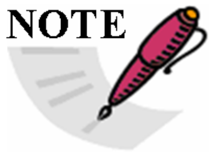
The layout of the CSDW is shown in Figure 6-48. The **uPacketCnt** value indicates the number of separate 1394 data messages in the data packet.

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

Figure 6-48. Type 0x58 IEEE 1394 Data Format 1 CSDW.

Individual 1394 data messages follow the CSDW. The format of the intra-packet data header is shown in Figure 6-49. The **suIntrPckTime** value is an eight byte representation of time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7). 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. A Bus Reset indication.
b. An Arbitration Reset Gap indication (both even and odd).
c. A Subaction Gap indication.
d. A Cycle Start indication (both even and odd).



See IEEE 1394b-2002 Section 17.8 for more details about interpreting **uStatus**.

```
struct SuIEEE1394F1_Header
    {
    uint64_t    suIntraPckTime;          // 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 6-49. Type 0x59 IEEE 1394 Data Format 1 intra-packet data header.

The complete 1394 bus data message follows the intra-packet data header. The length of the 1394 data message is indicated in the intra-packet data header. 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.

6.5.37 Type 0x5A - 0x5F, IEEE 1394 Data, Format 2 - Format 7. Reserved for future use.

6.5.38 Type 0x60, Parallel Data, Format 0. Parallel Data, Format 0 packets are used to record data bits received from a discrete parallel data interface. The number of bits that comprise one data word can range from 2 to 128 bits in length. A parallel data packet can contain multiple parallel data words

The Digital Cartridge Recording System (DCRsi) recording method and digital data interface were 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 (ECL). A simplified depiction of the interface is shown in Figure 6-50.

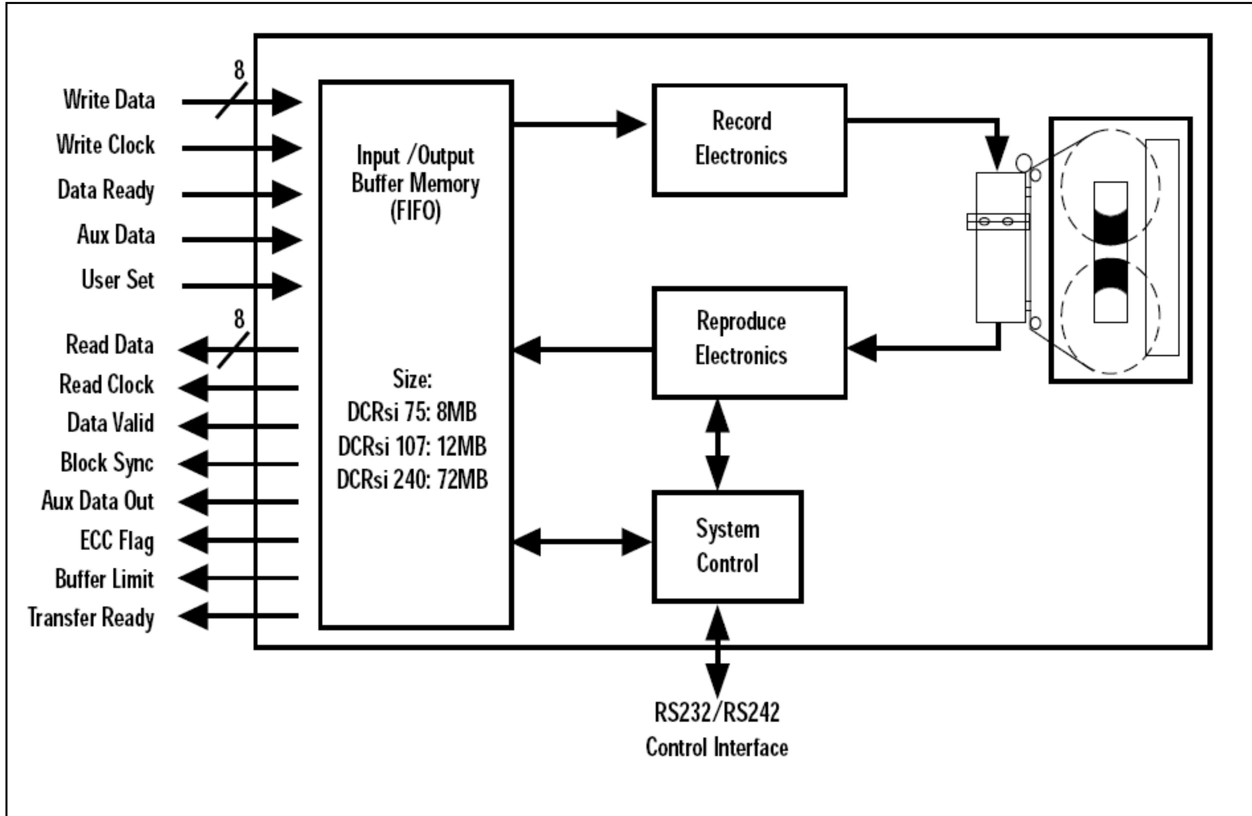


Figure 6-50. DCRsi interface.

The layout of the CSDW is shown in Figure 6-51. 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 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 6-51. Type 0x60 Parallel Data Format 0 CSDW.

Recorded parallel data follows the CSDW. There is no intra-packet header. 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 bytes per data word. See the Chapter 10 standard for complete details.

6.5.39 Type 0x61 - 0x67, Parallel Data, Format 1 - Format 7. Reserved for future use.

6.5.40 Type 0x68, Ethernet Data, Format 0. Ethernet Data, Format 0 packets are used to record Ethernet 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 6-52. The **iNumFrames** field indicates the number of Ethernet frames included in this data packet. The **iFormat** 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.

```
struct SuEthernetF0_ChanSpec
    {
    uint32_t    uNumFrames      : 16;     // Number of frames
    uint32_t    Reserved1       : 12;
    uint32_t    uFormat         :  4;     // Format of frames
    };
```

Figure 6-52. Type 0x68 Ethernet Data Format 0 CSDW.

Individual Ethernet data messages follow the CSDW. The format of the intra-packet data header is shown in Figure 6-53. The **suIntrPckTime** value is an eight byte representation of time in either 48-bit relative time format derived from the RTC (format shown in Figure 6-5) , or as absolute time. If this time is absolute time, it is in either Chapter 4 weighted 48-bit time (format shown in Figure 6-6) or IEEE 1588 time format (format shown in Figure 6-7). 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 **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. The **bFrameError** flag indicates an unspecified error has occurred in the reception of the Ethernet frame.

```
struct SuEthernetF0_Header
    {
    uint64_t    suIntraPckTime;             // Reference time
    uint32_t    uDataLen        : 14;       // Data length
    uint32_t    Reserved1       :  2;       //
    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    Reserved2       :  1;       //
    };
```

Figure 6-53. Type 0x68 Ethernet Data Format 0 intra-packet data header.

6.5.41 Type 0x69 - 0x6F, Ethernet Data, Format 1 - Format 7. Reserved for future use.

6.6 Time Interpretation

Chapter 10 defines a 48-bit Relative Time Counter (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; 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 a 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 nsec per tic = 15 msec).

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. For example, there may be separate time channels for time derived from IRIG B, Global Positioning System (GPS), and an internal battery backed up clock. In this scenario, all of these time sources are present in the data file as separate channels, each correlating the RTC to its own notion of clock time. The software application may allow the user to select which source of time to use for a given analysis. Alternatively, the software may decide the “best” source of time, depending on which time channels are providing valid time. In general, each time source will provide a slightly (or not so slightly) different clock time. It is usually most correct to select one time channel only and to use this channel exclusively to correlate RTC time to absolute clock time for all data packet types.

The stability of the RTC isn’t specified in Chapter 10 other than to require it to be at least as good as a common crystal oscillator. A good grade crystal oscillator can provide stability on the order of 10 ppm. Some vendors provide a RTC source considerably more stable than this. It

is tempting for an application to find a single time packet early in a recording and to use those time values to subsequently derive clock time from relative time. It is better to use the clock and relative time values from a time packet that occurs near the current data packet as the data file is decoded since there is some drift in the RTC during a recording session. It also may be the case that there is a jump in input clock time during a recording, such as when GPS locks for the first time, or when an IRIG time source is reprogrammed.

6.7 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. A general algorithm for reading all root and node index packets is as follows:

1. If "R-x\IDX\E" is 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,
    - Read the Node Index offset value
    - Move the read pointer to the Node Index offset value
    - Read the Node Index packet
    - Get the node index count from the CSDW
    - 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 the move read pointer to the next Root Index packet offset value
8. Read the next Root Index packet.
9. Go to Step 4.

6.8 Data Streaming

Chapter 10 recorders can stream their data over one of their download interface network ports using User Datagram Protocol (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 broadcast address. Different channels can be addressed to different addresses.

It is common to publish different groups of data to different multicast groups. According to RFC 3171, 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 2365, 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.

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 packets which are not required. 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. 16 multicast addresses is a common hardware limit. If a workstation needs to subscribe to more multicast addresses than the Ethernet hardware provides for, 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 implementation treat this as a signed value with a maximum value of 2^15 or 32768. Because of this, the maximum size of a Chapter 10 streaming packet should be no more than 32724 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 32k maximum size will prepend the non-segmented UDP transfer header shown in Figure 6-54. A Chapter 10 data packet larger than the 32k maximum size will need to be segmented before transmission, and will prepend the segmented UDP transfer header shown in Figure 6-55. IPv6 supports large data packets, negating the need for segmented data packets.

```
struct SuUdpTransferHeaderNonseg
    {
    uint32_t    uVersion        :  4;      // Version
    uint32_t    uType           :  4;      // Type of message
    uint32_t    uUdpSeqNum      : 24;      // UDP sequence number
    };
```

Figure 6-54. UDP Transfer Header, non-segmented data.

```
struct SuUdpTransferHeaderSeg
    {
    uint32_t    uVersion        :  4;      // Version
    uint32_t    uType           :  4;      // Type of message
    uint32_t    uUdpSeqNum      : 24;      // UDP sequence number
    uint32_t    uChanID         : 16;      // Channel ID
    uint32_t    uChanSeqNum     :  8;      // Channel sequence number
    uint32_t    uReserved       :  8;      //
    uint32_t    uSegOffset;                // Segment offset
    };
```

Figure 6-55. UDP Transfer Header, 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.

This page intentionally left blank.

CHAPTER 7

CONFORMANCE TO IRIG 106

Chapter 10 of the IRIG 106 standard sets forth requirements for digital recorders. Paragraph 10.3.1 summarizes requirements for a recorder to be considered 100 percent compliant with the standard. A number of features described in Chapter 10 are considered optional. Optional features are not required to be implemented; however, if they are implemented, they must be in accordance with the Chapter 10 standard.

Rather than reiterate all the requirements of Chapter 10, Table 7-1 and Table 7-2 present a brief outline of the major requirement areas, as well as those portions of Chapter 10 that have been identified as optional.

TABLE 7-1. PHYSICAL INTERFACE REQUIREMENTS

Physical Interfaces			
Ch 10 Section	Required	Optional	
10.3, 10.4.1	See Note		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	See Note		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	See Note		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		✔	Removable Memory Media (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

Note: 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 7-2. 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		✔	Intrapacket headers and time stamps

This page intentionally left blank.

APPENDIX A

SELECTED SOURCE CODE FILES

A software implementation of the IRIG 106 Chapter 9 and Chapter 10 standard is available from http:\\www.irig106.org; the software is open source, meaning it is freely available in source code form. It is written in ANSI C, and can be compiled in GNU Compiler Collection (GCC) as well as the various Microsoft Visual Studio compiler suites. Listed 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 http:\\www.irig106.org for the latest version of this source code as well as additional source code files.

The following source files are shown in the remaining portions of Appendix A.

Appendix	File	Comments
A-1 A-2	irig106ch10.h irig106ch10.c	Structures, macro definitions, and procedures for opening, reading/writing, and moving through an IRIG 106 data file.
A-3 A-4	i106_time.h i106_time.c	Structures, macro definitions, and procedures for handling time in general
A-5 A-6	i106_decode_time.h i106_decode_time.c	Structures, macro definitions, and procedures for decoding Time Data packets
A-7 A-8	i106_decode_tmats.h i106_decode_tmats.c	Structures, macro definitions, and procedures for decoding Computer Generated Data, Format 1 (TMATS Setup Record) packets.
A-9	config.h	Structures and macro definitions to provide portability across supported compiler suites.
A-10	stdint.h	Macro definitions to provide standard integer types of specific sizes across supported compiler suites.

This page intentionally left blank.

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.

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

#ifndef _irig106ch10_h_
#define _irig106ch10_h_

#ifdef __cplusplus
extern "C" {
#endif

#include "config.h"

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


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

#define MAX_HANDLES         4

#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_OVERFLOW       (uint8_t)0x10
#define I106CH10_PFLAGS_TIMESYNCERR    (uint8_t)0x20
#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_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    // Deprecated
#define I106CH10_DTYPE_IRIG_TIME       (uint8_t)0x11
#define I106CH10_DTYPE_1553_FMT_1      (uint8_t)0x19
#define I106CH10_DTYPE_1553_FMT_2      (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       (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_IMAGE_FMT_0     (uint8_t)0x48
#define I106CH10_DTYPE_IMAGE_FMT_1     (uint8_t)0x49
#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

/// Error return codes
typedef enum
    {
    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,   //
    I106_SEEK_ERROR          =  8,
```

```
    I106_WRONG_FILE_MODE     =  9,
    I106_NOT_OPEN            = 10,
    I106_ALREADY_OPEN        = 11,
    I106_BUFFER_TOO_SMALL    = 12,
    I106_NO_MORE_DATA        = 13,
    I106_NO_FREE_HANDLES     = 14,
    I106_INVALID_HANDLE      = 15,
    I106_TIME_NOT_FOUND      = 16,
    I106_HEADER_CHKSUM_BAD   = 17,
    I106_NO_INDEX            = 18,
    I106_UNSUPPORTED         = 19,
    } EnI106Status;

/// Data file open mode
typedef enum
    {
    I106_READ                = 1,    // Open existing file for reading
    I106_OVERWRITE           = 2,    // Create 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 existing file for reading in time order
    } EnI106Ch10Mode;

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

/// Index sort state
typedef enum
    {
    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
    uint32_t      aulTime[2];           ///< 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;
    int64_t     llTime;
    } SuFileIndex;

// Various file index array indexes
typedef struct
    {
    EnSortStatus    enSortStatus;
    SuFileIndex   * asuIndex;
    int             iArraySize;
    int             iArrayUsed;
    int             iArrayCurr;  // Current position in index array
    int64_t         llNextReadOffset;
    int             iNumSearchSteps;
    } SuIndex;


/// Data structure for IRIG 106 read/write handle
typedef struct
    {
    int             bInUse;
    int             iFile;
    char            szFileName[MAX_PATH];
    EnI106Ch10Mode  enFileMode;
    EnFileState     enFileState;
    SuIndex         suIndex;
    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[4];


/*
```

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

// Open / Close

EnI106Status I106_CALL_DECL
    enI106Ch10Open          (int                * piI106Ch10Handle,
                             const char           szOpenFileName[],
                             EnI106Ch10Mode       enMode);

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
    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);

uint32_t I106_CALL_DECL
    uCalcDataBuffReqSize(uint32_t uDataLen, int iChecksumType);

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

// In-order indexing
// -----------------

void I106_CALL_DECL
    vMakeInOrderIndex(int iHandle);

int I106_CALL_DECL
    bReadInOrderIndex(int iHandle, char * szIdxFileName);

int I106_CALL_DECL
    bWriteInOrderIndex(int iHandle, char * szIdxFileName);

#ifdef __cplusplus
}
#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/types.h>
#include <sys/stat.h>

#if defined(_WIN32)
#include <io.h>
#endif


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

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

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

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

/*
struct SuInOrderHdrInfo
    {
    SuI106Ch10Header            suHdr;
    struct SuInOrderHdrInfo   * psuNext;
    struct SuInOrderHdrInfo   * psuPrev;
    };
*/

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

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

// In order read linked list pointers
/*
struct SuInOrderHdrInfo   * m_psuFirstInOrderHdr  = NULL;
struct SuInOrderHdrInfo   * m_psuLastInOrderHdr   = NULL;
struct SuInOrderHdrInfo   * m_psuCurrInOrderHdr   = NULL;

struct SuInOrderHdrInfo   * m_psuFirstInOrderFree = NULL;
*/

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

#ifdef LOOK_AHEAD
void vCheckFillLookAheadBuffer(int iHandle);
#endif

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

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

    // Initialize handle data if necessary
    if (m_bHandlesInited == bFALSE)
        {
        for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
            g_suI106Handle[iIdx].bInUse = bFALSE;
        m_bHandlesInited = bTRUE;
        } // end if file handles not inited yet
```

```
    // Get the next available handle
    *piHandle = -1;
    for (iIdx=0; iIdx<MAX_HANDLES; iIdx++)
        {
        if (g_suI106Handle[iIdx].bInUse == bFALSE)
            {
            g_suI106Handle[iIdx].bInUse = bTRUE;
            *piHandle = iIdx;
            break;
            } // end if unused handle found
        } // end looking for unused handle

    if (*piHandle == -1)
        {
        return I106_NO_FREE_HANDLES;
        } // end if handle not found

    // Initialize some data
    g_suI106Handle[*piHandle].enFileState = enClosed;
    g_suI106Handle[*piHandle].suIndex.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(__GCC__)
       iFlags = O_RDONLY | O_LARGEFILE;
#else
       iFlags = O_RDONLY;
#endif
        g_suI106Handle[*piHandle].iFile = open(szFileName, iFlags, 0);
        if (g_suI106Handle[*piHandle].iFile == -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);
            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);
            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].enFileState = enReadHeader;

        // Make sure first packet is a config packet
        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;


        // 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;

        if (I106_READ_IN_ORDER == enMode)
            g_suI106Handle[*piHandle].suIndex.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_BINARY;
#elif defined(__GCC__)
           iFlags = O_WRONLY | O_CREAT | O_LARGEFILE;
#else
           iFlags = O_WRONLY | O_CREAT;
#endif
        g_suI106Handle[*piHandle].iFile = open(szFileName, iFlags, _S_IREAD | _S_IWRITE);
        if (g_suI106Handle[*piHandle].iFile == -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
        {
        return I106_OPEN_ERROR;
        }

    return I106_OK;
```

```
    }



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

EnI106Status I106_CALL_DECL
    enI106Ch10Close(int iHandle)
    {

    // If handles have 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;
        }

    // Make sure the file is really open
    if ((g_suI106Handle[iHandle].iFile   != -1) &&
        (g_suI106Handle[iHandle].bInUse  == bTRUE))
        {
        // Close the file
        close(g_suI106Handle[iHandle].iFile);
        }

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

    // Reset some status variables
    g_suI106Handle[iHandle].iFile       = -1;
    g_suI106Handle[iHandle].bInUse      = bFALSE;
    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 :
            enStatus = enI106Ch10ReadNextHeaderFile(iHandle, psuHeader);
```

```
            break;

        case I106_READ_IN_ORDER :
            if (g_suI106Handle[iHandle].suIndex.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 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;
            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;

            break;

        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.
    bReadHeaderWasOK = bTRUE;
    while (bTRUE)
        {

        // Read the header
        iReadCnt = read(g_suI106Handle[iHandle].iFile, psuHeader, HEADER_SIZE);

        // 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
            iReadCnt = read(g_suI106Handle[iHandle].iFile,
                            &psuHeader->aulTime[0], SEC_HEADER_SIZE);

            // 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 != 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->uChecksum != 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 4 bytes beyond previous read point
    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;

    } // 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)
    {

    SuIndex           * psuIndex = &g_suI106Handle[iHandle].suIndex;
    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                 bFound;
int                 iReadCnt;
int64_t             llSkipSize;
int64_t             llCurrPos;
EnI106Status        enStatus;

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

// 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.
        llSkipSize = g_suI106Handle[iHandle].ulCurrHeaderBuffLen +
                     g_suI106Handle[iHandle].ulCurrDataBuffReadPos;

        // Now to save some time backup more, at least the size of a header with no data
        llSkipSize += HEADER_SIZE;

        break;

    case enReadUnsynced :
        llSkipSize = 4;

        break;
    } // end switch file state


// Figure out where we're at and where in the file we want to be next
enI106Ch10GetPos(iHandle, &llCurrPos);

// If unsynced then make sure we are on a 4 byte offset
if (g_suI106Handle[iHandle].enFileState == enReadUnsynced)
    llSkipSize = llSkipSize - (llCurrPos % 4);

llCurrPos -= llSkipSize;

// Now loop forever looking for a valid packet or die trying
bFound = bFALSE;
while (bTRUE)
    {
    // Go to the new position and look for a legal header
    enStatus = enI106Ch10SetPos(iHandle, llCurrPos);
    if (enStatus != I106_OK)
```

```
            return I106_SEEK_ERROR;

        // Read and check the header sync
        iReadCnt = read(g_suI106Handle[iHandle].iFile, &(psuHeader->uSync), 2);
        if (iReadCnt != 2)
            return I106_SEEK_ERROR;
        if (psuHeader->uSync != IRIG106_SYNC)
            continue;

        // Sync pattern matched so check the header checksum
        iReadCnt = read(g_suI106Handle[iHandle].iFile, &(psuHeader->uChID), HEADER_SIZE-2);
        if (iReadCnt != HEADER_SIZE-2)
            return I106_SEEK_ERROR;
        if (psuHeader->uChecksum == uCalcHeaderChecksum(psuHeader))
            {
            bFound = bTRUE;
            break;
            }

        // No match, go back 4 more bytes and try again
        llCurrPos -= 4;

        // Check for begining of file
        if (llCurrPos < 0)
            {
            return I106_BOF;
            }

        } // end looping forever

    // If good header found then go back to just before the header
    // and call GetNextHeader() to let it do all the heavy lifting.
    if (bFound == bTRUE)
        {
        enStatus = enI106Ch10SetPos(iHandle, llCurrPos);
        if (enStatus != I106_OK)
            return I106_SEEK_ERROR;
        g_suI106Handle[iHandle].enFileState = enReadHeader;
        enStatus = enI106Ch10ReadNextHeader(iHandle, psuHeader);
        }

    return enStatus;
    } // end enI106Ch10ReadPrevHeader()



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

EnI106Status I106_CALL_DECL
    enI106Ch10ReadData(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 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
    iReadCnt = read(g_suI106Handle[iHandle].iFile, pvBuff, ulReadAmount);

    // 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;
        }

    // 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

    return I106_OK;
    }



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

EnI106Status I106_CALL_DECL
    enI106Ch10FirstMsg(int iHandle)
    {

    if (g_suI106Handle[iHandle].enFileMode == I106_READ_IN_ORDER)
        g_suI106Handle[iHandle].suIndex.iArrayCurr = 0;

    enI106Ch10SetPos(iHandle, 0L);
    return I106_OK;
    }



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

EnI106Status I106_CALL_DECL
    enI106Ch10LastMsg(int iHandle)
    {
    EnI106Status        enReturnStatus;
    EnI106Status        enStatus;
    int64_t             llPos;
    SuI106Ch10Header    suHeader;
```

```
    int                 iReadCnt;
    struct stat         suStatBuff;

    // If its opened for reading in order then just set the index pointer
    // to the last index.
    if (g_suI106Handle[iHandle].enFileMode == I106_READ_IN_ORDER)
        {
        g_suI106Handle[iHandle].suIndex.iArrayCurr =
            g_suI106Handle[iHandle].suIndex.iArrayUsed-1;
        enReturnStatus = I106_OK;
        }

    // If there is no index then do it the hard way
    else
        {

//      enReturnStatus = I106_SEEK_ERROR;

// MAYBE ALL WE NEED TO DO TO SEEK TO JUST PAST THE END, SET UNSYNC'ED STATE,
// AND THEN CALL enI106Ch10PrevMsg()

        // Figure out how big the file is and go to the end
//      llPos = filelength(_fileno(g_suI106Handle[iHandle].pFile)) - HEADER_SIZE;
        fstat(g_suI106Handle[iHandle].iFile, &suStatBuff);
        llPos = suStatBuff.st_size - HEADER_SIZE;

        //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
            if (llPos <= 0)
                {
                enReturnStatus = I106_SEEK_ERROR;
                break;
                }
```

```
            } // end looping forever
        } // end if not read in order mode

    // Go back to the good position
    enStatus = enI106Ch10SetPos(iHandle, llPos);

    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;
        }

    // Seek
#if defined(_MSC_VER)
    {
    __int64  llStatus;
    llStatus = _lseeki64(g_suI106Handle[iHandle].iFile, llOffset, SEEK_SET);
    }
#else
    lseek(g_suI106Handle[iHandle].iFile, llOffset, SEEK_SET);
#endif

    // Can't be sure we're on a message boundary so set unsync'ed
    g_suI106Handle[iHandle].enFileState = enReadUnsynced;

    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;
        }

    // Get position
#if defined(_MSC_VER)
    *pllOffset = _telli64(g_suI106Handle[iHandle].iFile);
#else
    *pllOffset = lseek(g_suI106Handle[iHandle].iFile, 0, SEEK_CUR);
#endif
```

```
    return I106_OK;
    }



/* -----------------------------------------------------------------------
 * Utilities
 * ----------------------------------------------------------------------- */

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;
    psuHeader->ubySeqNum      = uSeqNum;
    psuHeader->ubyPacketFlags = uFlags;
    psuHeader->ubyDataType    = uDataType;
    memset(&(psuHeader->aubyRefTime), 0, 6);
    psuHeader->uChecksum      = uCalcHeaderChecksum(psuHeader);
    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;
    }



// -----------------------------------------------------------------------
// Generate an index from the data file
// -----------------------------------------------------------------------

/*
Support for read back in time order is experimental.  Some 106-04 recorders
recorder data *way* out of time order.  But most others don't.  And starting
with 106-05 the most out of order is 1 second.

The best way to support read back in order is to do it on the fly as the file
is being read.  But that's more than I'm willing to do right now.  This indexing
scheme does get the job done for now.
*/

// Read the index from a previously generated index file.

int I106_CALL_DECL
    bReadInOrderIndex(int iHandle, char * szIdxFileName)
    {
    int                 iIdxFile;
    int                 iFlags;
    int                 iArrayReadStart;
    int                 iReadCnt;
```

```
    int                 bReadOK = bFALSE;
    SuIndex           * psuIndex = &g_suI106Handle[iHandle].suIndex;

    // Setup a one time loop to make it easy to break out on errors
    do
        {

        // Try opening and reading the index file
#if defined(_MSC_VER)
           iFlags = O_RDONLY | O_BINARY;
#else
           iFlags = O_RDONLY;
#endif
        iIdxFile = open(szIdxFileName, iFlags, 0);
        if (iIdxFile == -1)
            break;

        // Read the index data from the file
        while (bTRUE)
            {
            iArrayReadStart = psuIndex->iArraySize;
            psuIndex->iArraySize += 100;
            psuIndex->asuIndex = (SuFileIndex *)realloc(psuIndex->asuIndex,
                sizeof(SuFileIndex)*psuIndex->iArraySize);
            iReadCnt = read(iIdxFile, &(psuIndex->asuIndex[iArrayReadStart]),
                            100*sizeof(SuFileIndex));
            psuIndex->iArrayUsed += iReadCnt / sizeof(SuFileIndex);
            if (iReadCnt != 100*sizeof(SuFileIndex))
                break;
            } // end while reading data from file

        close(iIdxFile);

        // MIGHT WANT TO DO SOME SANITY CHECKS IN HERE

        psuIndex->enSortStatus = enSorted;
        bReadOK = bTRUE;
        } while (bFALSE); // end one time loop to read

    return bReadOK;
    }



// -----------------------------------------------------------------------

int I106_CALL_DECL
    bWriteInOrderIndex(int iHandle, char * szIdxFileName)
    {
    int                 iFlags;
    int                 iIdxFile;
    int                 iWriteIdx;
    SuIndex           * psuIndex = &g_suI106Handle[iHandle].suIndex;

    // Write out an index file for use next time
#if defined(_MSC_VER)
       iFlags = O_WRONLY | O_CREAT | O_BINARY;
#else
       iFlags = O_WRONLY | O_CREAT;
#endif
    iIdxFile = open(szIdxFileName, iFlags, _S_IREAD | _S_IWRITE);
    if (iIdxFile != -1)
        {
```

```
        // Read the index data from the file
        for (iWriteIdx=0; iWriteIdx<psuIndex->iArrayUsed; iWriteIdx++)
            {
            write(iIdxFile, &(psuIndex->asuIndex[iWriteIdx]), sizeof(SuFileIndex));
            } // end for all index array elements

        close(iIdxFile);
        }

    return bFALSE;
    }



// -----------------------------------------------------------------------

// This is used in qsort in vMakeInOrderIndex() below

int FileTimeCompare(const void * psuIndex1, const void * psuIndex2)
    {
    if (((SuFileIndex *)psuIndex1)->llTime < ((SuFileIndex *)psuIndex2)->llTime) return -1;
    if (((SuFileIndex *)psuIndex1)->llTime > ((SuFileIndex *)psuIndex2)->llTime) return  1;
    return 0;
    }


// -----------------------------------------------------------------------

// Read all headers and make an index based on time

void I106_CALL_DECL
    vMakeInOrderIndex(int iHandle)
    {

    EnI106Status        enStatus;
    int64_t             llStartPos;     // File position coming in
    int64_t             llCurrPos;      // Current file position
    SuI106Ch10Header    suHdr;          // Data packet header
    int64_t             llCurrTime;     // Current header time
    SuIndex           * psuIndex = &g_suI106Handle[iHandle].suIndex;

    // Remember the current file position
    enStatus = enI106Ch10GetPos(iHandle, &llStartPos);

    enStatus = enI106Ch10SetPos(iHandle, 0L);

    // Read headers, put time and file offset into index array
    while (bTRUE)
        {
        enStatus = enI106Ch10ReadNextHeaderFile(iHandle, &suHdr);

        // If EOF break out
        if (enStatus == I106_EOF)
            break;

        // If an error then clean up and get out
        if (enStatus != I106_OK)
            {
            free(psuIndex->asuIndex);
            psuIndex->asuIndex        = NULL;
            psuIndex->iArraySize      = 0;
            psuIndex->iArrayUsed      = 0;
```

```
            psuIndex->iNumSearchSteps = 0;
            psuIndex->enSortStatus    = enSortError;
            break;
            }

        // Get the time and position
        enStatus = enI106Ch10GetPos(iHandle, &llCurrPos);
        llCurrPos -= iGetHeaderLen(&suHdr);
        vTimeArray2LLInt(suHdr.aubyRefTime, &llCurrTime);

        // Check the array size, make it bigger if necessary
        if (psuIndex->iArrayUsed >= psuIndex->iArraySize)
            {
            psuIndex->iArraySize += 100;
            psuIndex->asuIndex =
                (SuFileIndex *)realloc(psuIndex->asuIndex,
                sizeof(SuFileIndex)*psuIndex->iArraySize);
            }

        // Copy the info into the next array element
        psuIndex->asuIndex[psuIndex->iArrayUsed].llOffset = llCurrPos;
        psuIndex->asuIndex[psuIndex->iArrayUsed].llTime   = llCurrTime;
        psuIndex->iArrayUsed++;
        }

    // Sort the index array
    // It is required that TMATS is the first record and IRIG time is the
    // second record so don't include those in the sort
    qsort(&(psuIndex->asuIndex[2]), psuIndex->iArrayUsed-2,
          sizeof(SuFileIndex), FileTimeCompare);

    // Put the file point back where we started and find the current index
// THIS SHOULD REALLY BE DONE FOR THE FILE-READ-OK LOGIC PATH ALSO
    enStatus = enI106Ch10SetPos(iHandle, llStartPos);
    psuIndex->iArrayCurr = 0;
    while (psuIndex->iArrayCurr < psuIndex->iArrayUsed)
        {
        if (llStartPos == psuIndex->asuIndex[psuIndex->iArrayCurr].llOffset)
            break;
        psuIndex->iArrayCurr++;
        }

    // If we didn't find it then it's an error
    if (psuIndex->iArrayCurr == psuIndex->iArrayUsed)
        {
        free(psuIndex->asuIndex);
        psuIndex->asuIndex        = NULL;
        psuIndex->iArraySize      = 0;
        psuIndex->iArrayUsed      = 0;
        psuIndex->iNumSearchSteps = 0;
        psuIndex->enSortStatus    = enSortError;
        }
    else
        psuIndex->enSortStatus = enSorted;

    return;
    }
```

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"

#ifdef __cplusplus
extern "C" {
#endif

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

typedef enum
    {
    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 struct
    {
    uint32_t        ulSecs;     // This is a time_t
    uint32_t        ulFrac;     // LSB = 100ns
    EnI106DateFmt   enFmt;      // Day or DMY format
    } SuIrig106Time;


// Relative time to absolute time reference
typedef struct
    {
    uint64_t        uRelTime;          // Relative time from header
    SuIrig106Time   suIrigTime;        // Clock time from IRIG source
    } 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


/// 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


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


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

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_Irig2RelTime(int               iI106Ch10Handle,
                        SuIrig106Time   * psuTime,
                        uint8_t           abyRelTime[]);

// 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);

// IT WOULD BE NICE TO HAVE SOME FUNCTIONS TO COMPARE 6 BYTE
// TIME ARRAY VALUES FOR EQUALITY AND INEQUALITY

// This is handy enough that we'll go ahead and export it to the world
// HMMM... MAYBE A BETTER WAY TO DO THIS IS TO MAKE THE TIME VARIABLES
// AND STRUCTURES THOSE DEFINED IN THIS PACKAGE.
time_t I106_CALL_DECL
    mkgmtime(struct tm * psuTmTime);

#ifdef __cplusplus
}
#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"

/*
 * 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)
    {
    uint64_t        uRelTime;
    int64_t         uTimeDiff;
    int64_t         lFracDiff;
    int64_t         lSecDiff;

    int64_t         lSec;
    int64_t         lFrac;

    uRelTime = 0L;
    memcpy(&uRelTime, &abyRelTime[0], 6);

    // Figure out the relative time difference
    uTimeDiff = uRelTime - 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;

    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;
    }



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

// 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,
                    int     iTimeLimit)
    {
    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;

    psuChanSpecTime = (SuTimeF1_ChanSpec *)pvBuff;

    // Get and save the current file position
    enStatus = enI106Ch10GetPos(iI106Ch10Handle, &llCurrOffset);
    if (enStatus != I106_OK)
        return enStatus;

    // Read the first 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);
                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;
    SuIndex           * psuIndex = &g_suI106Handle[iHandle].suIndex;
    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)
        {
        // Day / Month / Year format ("001:12:34:56.789")
        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 ("2008/02/29 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;
    }
```

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

/* 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))


time_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 (time_t)-1;

return (time_t)(86400L * days  + 3600L * hours + 60L * minutes + seconds);
}
```

This page intentionally left blank.

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
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;
```

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

/* Time Format 1 */

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

// Channel specific header
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  : 16;
#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

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


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

EnI106Status I106_CALL_DECL
    enI106_Decode_TimeF1(SuI106Ch10Header  * psuHeader,
                         void              * pvBuff,
                         SuIrig106Time     * psuTime);

EnI106Status I106_CALL_DECL
    enI106_Encode_TimeF1(SuI106Ch10Header  * psuHeader,
                         unsigned int        uExtTime,
                         unsigned int        uFmtTime,
                         unsigned int        uFmtDate,
                         SuIrig106Time     * psuTime,
                         void              * pvBuffTimeF1);

#ifdef __cplusplus
}
#endif

#endif
```

This page intentionally left blank.

APPENDIX A-6 - I106_DECODE_TIME.C

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

 i106_decode_time.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 <time.h>
#include <sys/types.h>
#include <sys/timeb.h>

#if defined(_WIN32)
#include <windows.h>        // For FILETIME
#endif

#include "stdint.h"

#include "irig106ch10.h"
#include "i106_time.h"
#include "i106_decode_time.h"


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

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

// Day of Year to Day and Month
typedef struct
    {
    int  iMonth;      // Month 0 - 11
    int  iDay;        // Day of month 1-31
    } SuDOY2DM;

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

// SuTimeRef   m_suCurrRefTime;           // Current value of IRIG reference time


// These structures are used to convert from day of the year format to
// day and month.  One is for normal years and the other is for leap years.
// The values and index are of the "struct tm" notion.  That is, the day of
// the year index is number of days since Jan 1st, i.e. Jan 1st = 0.  For
// IRIG time, Jan 1st = 1.  The month value is months since January, i.e.
// Jan = 0.  Don't get confused!

SuDOY2DM suDoy2DmNormal[] = {
{ 0,  1}, { 0,  2}, { 0,  3}, { 0,  4}, { 0,  5}, { 0,  6}, { 0,  7}, { 0,  8},
{ 0,  9}, { 0, 10}, { 0, 11}, { 0, 12}, { 0, 13}, { 0, 14}, { 0, 15}, { 0, 16},
{ 0, 17}, { 0, 18}, { 0, 19}, { 0, 20}, { 0, 21}, { 0, 22}, { 0, 23}, { 0, 24},
{ 0, 25}, { 0, 26}, { 0, 27}, { 0, 28}, { 0, 29}, { 0, 30}, { 0, 31}, { 1,  1},
{ 1,  2}, { 1,  3}, { 1,  4}, { 1,  5}, { 1,  6}, { 1,  7}, { 1,  8}, { 1,  9},
{ 1, 10}, { 1, 11}, { 1, 12}, { 1, 13}, { 1, 14}, { 1, 15}, { 1, 16}, { 1, 17},
{ 1, 18}, { 1, 19}, { 1, 20}, { 1, 21}, { 1, 22}, { 1, 23}, { 1, 24}, { 1, 25},
{ 1, 26}, { 1, 27}, { 1, 28}, { 2,  1}, { 2,  2}, { 2,  3}, { 2,  4}, { 2,  5},
{ 2,  6}, { 2,  7}, { 2,  8}, { 2,  9}, { 2, 10}, { 2, 11}, { 2, 12}, { 2, 13},
{ 2, 14}, { 2, 15}, { 2, 16}, { 2, 17}, { 2, 18}, { 2, 19}, { 2, 20}, { 2, 21},
{ 2, 22}, { 2, 23}, { 2, 24}, { 2, 25}, { 2, 26}, { 2, 27}, { 2, 28}, { 2, 29},
{ 2, 30}, { 2, 31}, { 3,  1}, { 3,  2}, { 3,  3}, { 3,  4}, { 3,  5}, { 3,  6},
{ 3,  7}, { 3,  8}, { 3,  9}, { 3, 10}, { 3, 11}, { 3, 12}, { 3, 13}, { 3, 14},
{ 3, 15}, { 3, 16}, { 3, 17}, { 3, 18}, { 3, 19}, { 3, 20}, { 3, 21}, { 3, 22},
{ 3, 23}, { 3, 24}, { 3, 25}, { 3, 26}, { 3, 27}, { 3, 28}, { 3, 29}, { 3, 30},
{ 4,  1}, { 4,  2}, { 4,  3}, { 4,  4}, { 4,  5}, { 4,  6}, { 4,  7}, { 4,  8},
{ 4,  9}, { 4, 10}, { 4, 11}, { 4, 12}, { 4, 13}, { 4, 14}, { 4, 15}, { 4, 16},
{ 4, 17}, { 4, 18}, { 4, 19}, { 4, 20}, { 4, 21}, { 4, 22}, { 4, 23}, { 4, 24},
{ 4, 25}, { 4, 26}, { 4, 27}, { 4, 28}, { 4, 29}, { 4, 30}, { 4, 31}, { 5,  1},
{ 5,  2}, { 5,  3}, { 5,  4}, { 5,  5}, { 5,  6}, { 5,  7}, { 5,  8}, { 5,  9},
{ 5, 10}, { 5, 11}, { 5, 12}, { 5, 13}, { 5, 14}, { 5, 15}, { 5, 16}, { 5, 17},
{ 5, 18}, { 5, 19}, { 5, 20}, { 5, 21}, { 5, 22}, { 5, 23}, { 5, 24}, { 5, 25},
{ 5, 26}, { 5, 27}, { 5, 28}, { 5, 29}, { 5, 30}, { 6,  1}, { 6,  2}, { 6,  3},
{ 6,  4}, { 6,  5}, { 6,  6}, { 6,  7}, { 6,  8}, { 6,  9}, { 6, 10}, { 6, 11},
{ 6, 12}, { 6, 13}, { 6, 14}, { 6, 15}, { 6, 16}, { 6, 17}, { 6, 18}, { 6, 19},
{ 6, 20}, { 6, 21}, { 6, 22}, { 6, 23}, { 6, 24}, { 6, 25}, { 6, 26}, { 6, 27},
{ 6, 28}, { 6, 29}, { 6, 30}, { 6, 31}, { 7,  1}, { 7,  2}, { 7,  3}, { 7,  4},
{ 7,  5}, { 7,  6}, { 7,  7}, { 7,  8}, { 7,  9}, { 7, 10}, { 7, 11}, { 7, 12},
{ 7, 13}, { 7, 14}, { 7, 15}, { 7, 16}, { 7, 17}, { 7, 18}, { 7, 19}, { 7, 20},
{ 7, 21}, { 7, 22}, { 7, 23}, { 7, 24}, { 7, 25}, { 7, 26}, { 7, 27}, { 7, 28},
{ 7, 29}, { 7, 30}, { 7, 31}, { 8,  1}, { 8,  2}, { 8,  3}, { 8,  4}, { 8,  5},
{ 8,  6}, { 8,  7}, { 8,  8}, { 8,  9}, { 8, 10}, { 8, 11}, { 8, 12}, { 8, 13},
{ 8, 14}, { 8, 15}, { 8, 16}, { 8, 17}, { 8, 18}, { 8, 19}, { 8, 20}, { 8, 21},
{ 8, 22}, { 8, 23}, { 8, 24}, { 8, 25}, { 8, 26}, { 8, 27}, { 8, 28}, { 8, 29},
{ 8, 30}, { 9,  1}, { 9,  2}, { 9,  3}, { 9,  4}, { 9,  5}, { 9,  6}, { 9,  7},
```

```
{ 9,  8}, { 9,  9}, { 9, 10}, { 9, 11}, { 9, 12}, { 9, 13}, { 9, 14}, { 9, 15},
{ 9, 16}, { 9, 17}, { 9, 18}, { 9, 19}, { 9, 20}, { 9, 21}, { 9, 22}, { 9, 23},
{ 9, 24}, { 9, 25}, { 9, 26}, { 9, 27}, { 9, 28}, { 9, 29}, { 9, 30}, { 9, 31},
{10,  1}, {10,  2}, {10,  3}, {10,  4}, {10,  5}, {10,  6}, {10,  7}, {10,  8},
{10,  9}, {10, 10}, {10, 11}, {10, 12}, {10, 13}, {10, 14}, {10, 15}, {10, 16},
{10, 17}, {10, 18}, {10, 19}, {10, 20}, {10, 21}, {10, 22}, {10, 23}, {10, 24},
{10, 25}, {10, 26}, {10, 27}, {10, 28}, {10, 29}, {10, 30}, {11,  1}, {11,  2},
{11,  3}, {11,  4}, {11,  5}, {11,  6}, {11,  7}, {11,  8}, {11,  9}, {11, 10},
{11, 11}, {11, 12}, {11, 13}, {11, 14}, {11, 15}, {11, 16}, {11, 17}, {11, 18},
{11, 19}, {11, 20}, {11, 21}, {11, 22}, {11, 23}, {11, 24}, {11, 25}, {11, 26},
{11, 27}, {11, 28}, {11, 29}, {11, 30}, {11, 31} };

SuDOY2DM suDoy2DmLeap[] = {
{ 0,  1}, { 0,  2}, { 0,  3}, { 0,  4}, { 0,  5}, { 0,  6}, { 0,  7}, { 0,  8},
{ 0,  9}, { 0, 10}, { 0, 11}, { 0, 12}, { 0, 13}, { 0, 14}, { 0, 15}, { 0, 16},
{ 0, 17}, { 0, 18}, { 0, 19}, { 0, 20}, { 0, 21}, { 0, 22}, { 0, 23}, { 0, 24},
{ 0, 25}, { 0, 26}, { 0, 27}, { 0, 28}, { 0, 29}, { 0, 30}, { 0, 31}, { 1,  1},
{ 1,  2}, { 1,  3}, { 1,  4}, { 1,  5}, { 1,  6}, { 1,  7}, { 1,  8}, { 1,  9},
{ 1, 10}, { 1, 11}, { 1, 12}, { 1, 13}, { 1, 14}, { 1, 15}, { 1, 16}, { 1, 17},
{ 1, 18}, { 1, 19}, { 1, 20}, { 1, 21}, { 1, 22}, { 1, 23}, { 1, 24}, { 1, 25},
{ 1, 26}, { 1, 27}, { 1, 28}, { 1, 29}, { 2,  1}, { 2,  2}, { 2,  3}, { 2,  4},
{ 2,  5}, { 2,  6}, { 2,  7}, { 2,  8}, { 2,  9}, { 2, 10}, { 2, 11}, { 2, 12},
{ 2, 13}, { 2, 14}, { 2, 15}, { 2, 16}, { 2, 17}, { 2, 18}, { 2, 19}, { 2, 20},
{ 2, 21}, { 2, 22}, { 2, 23}, { 2, 24}, { 2, 25}, { 2, 26}, { 2, 27}, { 2, 28},
{ 2, 29}, { 2, 30}, { 2, 31}, { 3,  1}, { 3,  2}, { 3,  3}, { 3,  4}, { 3,  5},
{ 3,  6}, { 3,  7}, { 3,  8}, { 3,  9}, { 3, 10}, { 3, 11}, { 3, 12}, { 3, 13},
{ 3, 14}, { 3, 15}, { 3, 16}, { 3, 17}, { 3, 18}, { 3, 19}, { 3, 20}, { 3, 21},
{ 3, 22}, { 3, 23}, { 3, 24}, { 3, 25}, { 3, 26}, { 3, 27}, { 3, 28}, { 3, 29},
{ 3, 30}, { 4,  1}, { 4,  2}, { 4,  3}, { 4,  4}, { 4,  5}, { 4,  6}, { 4,  7},
{ 4,  8}, { 4,  9}, { 4, 10}, { 4, 11}, { 4, 12}, { 4, 13}, { 4, 14}, { 4, 15},
{ 4, 16}, { 4, 17}, { 4, 18}, { 4, 19}, { 4, 20}, { 4, 21}, { 4, 22}, { 4, 23},
{ 4, 24}, { 4, 25}, { 4, 26}, { 4, 27}, { 4, 28}, { 4, 29}, { 4, 30}, { 4, 31},
{ 5,  1}, { 5,  2}, { 5,  3}, { 5,  4}, { 5,  5}, { 5,  6}, { 5,  7}, { 5,  8},
{ 5,  9}, { 5, 10}, { 5, 11}, { 5, 12}, { 5, 13}, { 5, 14}, { 5, 15}, { 5, 16},
{ 5, 17}, { 5, 18}, { 5, 19}, { 5, 20}, { 5, 21}, { 5, 22}, { 5, 23}, { 5, 24},
{ 5, 25}, { 5, 26}, { 5, 27}, { 5, 28}, { 5, 29}, { 5, 30}, { 6,  1}, { 6,  2},
{ 6,  3}, { 6,  4}, { 6,  5}, { 6,  6}, { 6,  7}, { 6,  8}, { 6,  9}, { 6, 10},
{ 6, 11}, { 6, 12}, { 6, 13}, { 6, 14}, { 6, 15}, { 6, 16}, { 6, 17}, { 6, 18},
{ 6, 19}, { 6, 20}, { 6, 21}, { 6, 22}, { 6, 23}, { 6, 24}, { 6, 25}, { 6, 26},
{ 6, 27}, { 6, 28}, { 6, 29}, { 6, 30}, { 6, 31}, { 7,  1}, { 7,  2}, { 7,  3},
{ 7,  4}, { 7,  5}, { 7,  6}, { 7,  7}, { 7,  8}, { 7,  9}, { 7, 10}, { 7, 11},
{ 7, 12}, { 7, 13}, { 7, 14}, { 7, 15}, { 7, 16}, { 7, 17}, { 7, 18}, { 7, 19},
{ 7, 20}, { 7, 21}, { 7, 22}, { 7, 23}, { 7, 24}, { 7, 25}, { 7, 26}, { 7, 27},
{ 7, 28}, { 7, 29}, { 7, 30}, { 7, 31}, { 8,  1}, { 8,  2}, { 8,  3}, { 8,  4},
{ 8,  5}, { 8,  6}, { 8,  7}, { 8,  8}, { 8,  9}, { 8, 10}, { 8, 11}, { 8, 12},
{ 8, 13}, { 8, 14}, { 8, 15}, { 8, 16}, { 8, 17}, { 8, 18}, { 8, 19}, { 8, 20},
{ 8, 21}, { 8, 22}, { 8, 23}, { 8, 24}, { 8, 25}, { 8, 26}, { 8, 27}, { 8, 28},
{ 8, 29}, { 8, 30}, { 9,  1}, { 9,  2}, { 9,  3}, { 9,  4}, { 9,  5}, { 9,  6},
{ 9,  7}, { 9,  8}, { 9,  9}, { 9, 10}, { 9, 11}, { 9, 12}, { 9, 13}, { 9, 14},
{ 9, 15}, { 9, 16}, { 9, 17}, { 9, 18}, { 9, 19}, { 9, 20}, { 9, 21}, { 9, 22},
{ 9, 23}, { 9, 24}, { 9, 25}, { 9, 26}, { 9, 27}, { 9, 28}, { 9, 29}, { 9, 30},
{ 9, 31}, {10,  1}, {10,  2}, {10,  3}, {10,  4}, {10,  5}, {10,  6}, {10,  7},
{10,  8}, {10,  9}, {10, 10}, {10, 11}, {10, 12}, {10, 13}, {10, 14}, {10, 15},
{10, 16}, {10, 17}, {10, 18}, {10, 19}, {10, 20}, {10, 21}, {10, 22}, {10, 23},
{10, 24}, {10, 25}, {10, 26}, {10, 27}, {10, 28}, {10, 29}, {10, 30}, {11,  1},
{11,  2}, {11,  3}, {11,  4}, {11,  5}, {11,  6}, {11,  7}, {11,  8}, {11,  9},
{11, 10}, {11, 11}, {11, 12}, {11, 13}, {11, 14}, {11, 15}, {11, 16}, {11, 17},
{11, 18}, {11, 19}, {11, 20}, {11, 21}, {11, 22}, {11, 23}, {11, 24}, {11, 25},
{11, 26}, {11, 27}, {11, 28}, {11, 29}, {11, 30}, {11, 31} };
```

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


/* ======================================================================= */

// Take an IRIG F1 time packet and decode it into something we can use

EnI106Status I106_CALL_DECL
    enI106_Decode_TimeF1(SuI106Ch10Header  * psuHeader,
                         void              * pvBuff,
                         SuIrig106Time     * psuTime)
    {
//    time_t                lTime;
    struct tm             suTmTime;
//    struct timeval        suTvTime;
    SuTimeF1_ChanSpec   * psuChanSpecTime;
    SuTime_MsgDmyFmt    * psuTimeDmy;
    SuTime_MsgDayFmt    * psuTimeDay;

    psuChanSpecTime = (SuTimeF1_ChanSpec *)pvBuff;

    // Time in Day format
// HMMMMM.... THIS ISN'T QUITE RIGHT. DID THE STANDARD CHANGE???
    if (psuChanSpecTime->uDateFmt == 0)
        {
        // Make
        psuTimeDay = (SuTime_MsgDayFmt *)((char *)pvBuff + sizeof(SuTimeF1_ChanSpec));
        suTmTime.tm_sec   = psuTimeDay->uTSn *  10 + psuTimeDay->uSn;
        suTmTime.tm_min   = psuTimeDay->uTMn *  10 + psuTimeDay->uMn;
        suTmTime.tm_hour  = psuTimeDay->uTHn *  10 + psuTimeDay->uHn;
        suTmTime.tm_yday  = psuTimeDay->uHDn * 100 + psuTimeDay->uTDn * 10 +
            psuTimeDay->uDn - 1;
        suTmTime.tm_mday  = suDoy2DmNormal[suTmTime.tm_yday].iDay;
        suTmTime.tm_mon   = suDoy2DmNormal[suTmTime.tm_yday].iMonth;
        suTmTime.tm_year  = 70;  // i.e. 1970
        suTmTime.tm_isdst = 0;
        psuTime->ulSecs   = mkgmtime(&suTmTime);
        psuTime->ulFrac   = psuTimeDay->uHmn * 1000000L + psuTimeDay->uTmn * 100000L;
        psuTime->enFmt    = I106_DATEFMT_DAY;
        }

    // Time in DMY format
    else
        {
        psuTimeDmy = (SuTime_MsgDmyFmt *)((char *)pvBuff + sizeof(SuTimeF1_ChanSpec));
        suTmTime.tm_sec   = psuTimeDmy->uTSn *   10 + psuTimeDmy->uSn;
        suTmTime.tm_min   = psuTimeDmy->uTMn *   10 + psuTimeDmy->uMn;
        suTmTime.tm_hour  = psuTimeDmy->uTHn *   10 + psuTimeDmy->uHn;
        suTmTime.tm_yday  = 0;
        suTmTime.tm_mday  = psuTimeDmy->uTDn *   10 + psuTimeDmy->uDn;
        suTmTime.tm_mon   = psuTimeDmy->uTOn *   10 + psuTimeDmy->uOn - 1;
        suTmTime.tm_year  = psuTimeDmy->uOYn * 1000 + psuTimeDmy->uHYn * 100 +
                            psuTimeDmy->uTYn *   10 + psuTimeDmy->uYn - 1900;
        suTmTime.tm_isdst = 0;
        psuTime->ulSecs   = mkgmtime(&suTmTime);
        psuTime->ulFrac   = psuTimeDmy->uHmn * 1000000L + psuTimeDmy->uTmn * 100000L;
        psuTime->enFmt    = I106_DATEFMT_DMY;
        }

    return I106_OK;
```

```
    }



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

  This function returns the day of the year that corresponds to the date in
  the input parameter dt.

 ------------------------------------------------------------------------- */
/*
static int iDay_Of_Year(struct date *ptDate)
{
  int         *paiLastDayOfMonth;

  // Figure out leap year
  if ((ptDate->da_year % 4) != 0) paiLastDayOfMonth = aiLastDayOfMonthNormal;
  else                            paiLastDayOfMonth = aiLastDayOfMonthLeapYear;

  return paiLastDayOfMonth[ptDate->da_mon-1] + ptDate->da_day;

}
*/


EnI106Status I106_CALL_DECL
    enI106_Encode_TimeF1(SuI106Ch10Header  * psuHeader,
                         unsigned int        uTimeSrc,
                         unsigned int        uFmtTime,
                         unsigned int        uFmtDate,
                         SuIrig106Time     * psuTime,
                         void              * pvBuffTimeF1)
    {
    // A temporary integer to decimate to get BCD factors
    uint32_t          uIntDec;
    struct tm       * psuTmTime;

    typedef struct
        {
        SuTimeF1_ChanSpec   suChanSpec;
        union
            {
            SuTime_MsgDayFmt    suDayFmt;
            SuTime_MsgDmyFmt    suDmyFmt;
            } suMsg;
        } SuMsgTimeF1;

    SuMsgTimeF1 * psuTimeF1;

    SuTime_MsgDayFmt   * psuDayFmt;
    SuTime_MsgDmyFmt   * psuDmyFmt;

    // Now, after creating this ubertime-structure above, create a
    // couple of pointers to make the code below simpler to read.
    psuTimeF1 = (SuMsgTimeF1 *)pvBuffTimeF1;
    psuDayFmt = &(psuTimeF1->suMsg.suDayFmt);
    psuDmyFmt = &(psuTimeF1->suMsg.suDmyFmt);

    // Zero out all the time fields
    memset(psuTimeF1, 0, sizeof(SuTimeF1_ChanSpec));

    // Break time down to DMY HMS
    psuTmTime = gmtime((time_t *)&(psuTime->ulSecs));
```

```
// Make channel specific data word
psuTimeF1->suChanSpec.uTimeSrc    = uTimeSrc;
psuTimeF1->suChanSpec.uTimeFmt    = uFmtTime;
psuTimeF1->suChanSpec.uDateFmt    = uFmtDate;
if (psuTmTime->tm_year % 4 == 0)
    psuTimeF1->suChanSpec.bLeapYear = 1;
else
    psuTimeF1->suChanSpec.bLeapYear = 0;

// Fill in day of year format
if (uFmtDate == 0)
    {
    // Zero out all the time fields
    memset(psuDayFmt, 0, sizeof(SuTime_MsgDayFmt));

    // Set the various time fields
    uIntDec = psuTime->ulFrac / 100000L;
    psuDayFmt->uTmn = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDayFmt->uHmn) / 10;
    psuDayFmt->uHmn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_sec;
    psuDayFmt->uSn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDayFmt->uSn)  / 10;
    psuDayFmt->uTSn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_min;
    psuDayFmt->uMn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDayFmt->uMn)  / 10;
    psuDayFmt->uTMn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_hour;
    psuDayFmt->uHn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDayFmt->uHn)  / 10;
    psuDayFmt->uTHn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_yday + 1;
    psuDayFmt->uDn = (uint16_t)(uIntDec   % 10);
    uIntDec = (uIntDec - psuDayFmt->uDn)  / 10;
    psuDayFmt->uTDn = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDayFmt->uTDn) / 10;
    psuDayFmt->uHDn  = (uint16_t)(uIntDec  % 10);

    // Set the data length in the header
    psuHeader->ulDataLen =
        sizeof(SuTimeF1_ChanSpec) + sizeof(SuTime_MsgDayFmt);
    }

// Fill in day, month, year format
else
    {
    // Zero out all the time fields
    memset(psuDmyFmt, 0, sizeof(SuTime_MsgDmyFmt));

    // Set the various time fields
    uIntDec = psuTime->ulFrac / 100000L;
    psuDmyFmt->uTmn = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uHmn) / 10;
    psuDmyFmt->uHmn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_sec;
    psuDmyFmt->uSn  = (uint16_t)(uIntDec  % 10);
```

```
    uIntDec = (uIntDec - psuDmyFmt->uSn)  / 10;
    psuDmyFmt->uTSn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_min;
    psuDmyFmt->uMn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uMn)  / 10;
    psuDmyFmt->uTMn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_hour;
    psuDmyFmt->uHn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uHn)  / 10;
    psuDmyFmt->uTHn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_mday;
    psuDmyFmt->uDn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uDn)  / 10;
    psuDmyFmt->uTDn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_mon + 1;
    psuDmyFmt->uOn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uOn)  / 10;
    psuDmyFmt->uTOn = (uint16_t)(uIntDec  % 10);

    uIntDec = psuTmTime->tm_year + 1900;
    psuDmyFmt->uYn  = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uYn)  / 10;
    psuDmyFmt->uTYn = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uTYn) / 10;
    psuDmyFmt->uHYn = (uint16_t)(uIntDec  % 10);
    uIntDec = (uIntDec - psuDmyFmt->uHYn) / 10;
    psuDmyFmt->uOYn = (uint16_t)(uIntDec  % 10);

    // Set the data length in the header
    psuHeader->ulDataLen =
        sizeof(SuTimeF1_ChanSpec) + sizeof(SuTime_MsgDmyFmt);
    }

// Make the data buffer checksum and update the header
uAddDataFillerChecksum(psuHeader, (unsigned char *)pvBuffTimeF1);

return I106_OK;
}
```

This page intentionally left blank.

APPENDIX A-7 - I106_DECODE_TMATS.H

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

 i106_decode_tmats.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_TMATS_H
#define _I106_DECODE_TMATS_H

#ifdef __cplusplus
extern "C" {
#endif

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


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

// Channel specific data word
// --------------------------

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

```
#endif

typedef PUBLIC struct Tmats_ChanSpec_S
    {
    uint32_t    iCh10Ver        :  8;      // Recorder Ch 10 Version
    uint32_t    bConfigChange   :  1;      // Recorder configuration changed
    uint32_t    iReserved       : 23;      // Reserved
#if !defined(__GNUC__)
    } SuTmats_ChanSpec;
#else
    } __attribute__ ((packed)) SuTmats_ChanSpec;
#endif

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


// B Records
// ---------

typedef PUBLIC struct SuBRecord_S
    {
    int                         iRecordNum;             // B-x
    char                      * szDataLinkName;         // B-x\DLN
    int                         iNumBuses;              // B-x\NBS\N
    struct SuBRecord_S        * psuNextBRecord;
    } SuBRecord;


// M Records
// ---------

typedef PUBLIC struct SuMRecord_S
    {
    int                         iRecordNum;             // M-x
    char                      * szDataSourceID;         // M-x\ID
    char                      * szDataLinkName;         // M-x\BB\DLN
    char                      * szBasebandSignalType;   // M-x\BSG1
    struct SuBRecord_S        * psuBRecord;             // Corresponding B record
    struct SuMRecord_S        * psuNextMRecord;         // Used to keep track of M records
    } SuMRecord;


// R Records
// ---------

// R record data source
typedef PUBLIC struct SuRDataSource_S
    {
    int                         iDataSourceNum;         // R-x\XXX-n
    char                      * szDataSourceID;         // R-x\DSI-n
    char                      * szChannelDataType;      // R-x\CDT-n
    int                         iTrackNumber;           // R-x\TK1-n
    int                         bEnabled;               // R-x\CHE-n
    struct SuMRecord_S        * psuMRecord;             // Corresponding M record
    struct SuRDataSource_S    * psuNextRDataSource;
    } SuRDataSource;

// R record
typedef PUBLIC struct SuRRecord_S
    {
    int                         iRecordNum;             // R-x
```

```
    char                       * szDataSourceID;         // R-x\ID
    int                          iNumDataSources;        // R-x\N
    SuRDataSource              * psuFirstDataSource;     //
    struct SuRRecord_S         * psuNextRRecord;         // Used to keep track of R records
    } SuRRecord;


// G Records
// ---------

// G record, data source
typedef PUBLIC struct SuGDataSource_S
    {
    int                          iDataSourceNum;         // G\XXX-n
    char                       * szDataSourceID;         // G\DSI-n
    char                       * szDataSourceType;       // G\DST-n
    struct SuRRecord_S         * psuRRecord;             // Corresponding R record
    struct SuGDataSource_S     * psuNextGDataSource;
    } SuGDataSource;

// G record
typedef PUBLIC struct GRecord_S
    {
    char                       * szProgramName;          // G\PN
    char                       * szIrig106Rev;           // G\106
    int                          iNumDataSources;        // G\DSI\N
    SuGDataSource              * psuFirstGDataSource;
    } SuGRecord;

// Memory linked list
// ------------------

// Linked list that keeps track of malloc'ed memory
typedef PUBLIC struct MemBlock_S
    {
    void                     * pvMemBlock;
    struct MemBlock_S        * psuNextMemBlock;
    } SuMemBlock;

// Decoded TMATS info
// ------------------

typedef PUBLIC struct SuTmatsInfo_S
    {
    int               iCh10Ver;
    int               bConfigChange;
    SuGRecord       * psuFirstGRecord;
    SuRRecord       * psuFirstRRecord;
    SuMRecord       * psuFirstMRecord;
    SuBRecord       * psuFirstBRecord;
    void            * psuFirstTRecord;
    void            * psuFirstPRecord;
    void            * psuFirstDRecord;
    void            * psuFirstSRecord;
    void            * psuFirstARecord;
    void            * psuFirstCRecord;
    void            * psuFirstHRecord;
    void            * psuFirstVRecord;
    SuMemBlock      * psuFirstMemBlock;
    } SuTmatsInfo;

/*
 * Function Declaration
```

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

EnI106Status I106_CALL_DECL
    enI106_Decode_Tmats(SuI106Ch10Header * psuHeader,
                        void             * pvBuff,
                        SuTmatsInfo      * psuTmatsInfo);

void I106_CALL_DECL
    enI106_Free_TmatsInfo(SuTmatsInfo    * psuTmatsInfo);

I106_CALL_DECL EnI106Status
    enI106_Encode_Tmats(SuI106Ch10Header * psuHeader,
                        void             * pvBuff,
                        char             * szTMATS);

#ifdef __cplusplus
}
#endif

#endif
```

APPENDIX A-8 - I106_DECODE_TMATS.C

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

 i106_decode_tmats.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 <malloc.h>
#include <assert.h>

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

#include "irig106ch10.h"
#include "i106_decode_tmats.h"


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

Here's how this module decodes and stores TMATS data.

Any field that is to be decoded and stored must have a corresponding entry
in one of the defined data structures.  In other words, there is no support
for storing and later retrieving arbitrary TMATS lines.  Maybe that would have
been better, but for now only TMATS lines that are understood by this module
will be stored.

This module makes no assumptions about the ordering, numbering, or
```

```
numbers of TMATS lines. Information is stored in linked lists that are
created and grow as needed.  For now there is the assumption that there
is only one G record, but there may be multiples of other records.

There is a linked list for each type of record (except the one and only
G record).  As the TMATS lines are read one by one, the info is decoded
and stored in the appropriate existing record, or a new record is create
if necessary.

After all TMATS lines are read and decoded, the linked lists are scanned
and connected into a tree.  That is, R records are connected to the
corresponding G, M records are connected to the corresponding R, etc.
When done, the TMATS info is organized into a tree similar to that depicted
in IRIG 106 Chapter 9 (TMATS) Figure 9-1 "Group relationships".

There are at least two ways to use this information.  One is to start with
the top level G record and walk the tree.  This is a good way to provide
access to all the decoded data, for example, to print out everything in
the tree.  The other way to access data is to start at the beginning of
one of the linked lists of records, and walk the linked list.  This might
be a good way to get just some specific data, like a list of Channel ID's
for 1553IN type data sources.

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


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

#define CR      (13)
#define LF      (10)

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

// 1553 bus attributes
// -------------------

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

// This is an empty string that text fields can point to before
// they get a value. This ensures that if fields don't get set while
// reading the TMATS record they will point to something benign.
char                    m_szEmpty[] = "";

static SuTmatsInfo      * m_psuTmatsInfo;

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

int bDecodeGLine(char * szCodeName, char * szDataItem, SuGRecord ** ppsuFirstGRec);
int bDecodeRLine(char * szCodeName, char * szDataItem, SuRRecord ** ppsuFirstRRec);
int bDecodeMLine(char * szCodeName, char * szDataItem, SuMRecord ** ppsuFirstMRec);
int bDecodeBLine(char * szCodeName, char * szDataItem, SuBRecord ** ppsuFirstBRec);
```

```
SuRRecord * psuGetRRecord(SuRRecord ** ppsuFirstRRec, int iRIndex, int bMakeNew);
SuMRecord * psuGetMRecord(SuMRecord ** ppsuFirstMRec, int iRIndex, int bMakeNew);
SuBRecord * psuGetBRecord(SuBRecord ** ppsuFirstBRec, int iRIndex, int bMakeNew);

SuGDataSource * psuGetGDataSource(SuGRecord * psuGRec, int iDSIIndex, int bMakeNew);
SuRDataSource * psuGetRDataSource(SuRRecord * psuRRec, int iDSIIndex, int bMakeNew);

void vConnectRtoG(SuGRecord * psuFirstGRecord, SuRRecord * psuFirstRRecord);
void vConnectMtoR(SuRRecord * psuFirstRRecord, SuMRecord * psuFirstMRecord);
void vConnectBtoM(SuMRecord * psuFirstMRecord, SuBRecord * psuFirstBRecord);

void * TmatsMalloc(size_t iSize);

/* ======================================================================== */

/* The idea behind this routine is to read the TMATS record, parse it, and
 * put the various data fields into a tree structure that can be used later
 * to find various settings.
 */

EnI106Status I106_CALL_DECL
    enI106_Decode_Tmats(SuI106Ch10Header * psuHeader,
                        void             * pvBuff,
                        SuTmatsInfo      * psuTmatsInfo)
    {
    unsigned long       iInBuffIdx;
    char              * achInBuff;
    char                szLine[2048];
    int                 iLineIdx;
    char              * szCodeName;
    char              * szDataItem;
    int                 bParseError;
    SuTmats_ChanSpec  * psuTmats_ChanSpec;

    // Store a copy for module wide use
    m_psuTmatsInfo = psuTmatsInfo;

    // Initialize the TMATS info data structure
    enI106_Free_TmatsInfo(psuTmatsInfo);

    // Decode any available info from channel specific data word
    switch (psuHeader->ubyHdrVer)
        {
        case 0x03 :  // 106-07
            psuTmats_ChanSpec = (SuTmats_ChanSpec *)pvBuff;
            psuTmatsInfo->iCh10Ver      = psuTmats_ChanSpec->iCh10Ver;
            psuTmatsInfo->bConfigChange = psuTmats_ChanSpec->bConfigChange;
            break;
        default :
            psuTmatsInfo->iCh10Ver      = 0x00;
            psuTmatsInfo->bConfigChange = 0x00;
            break;
        }

    // Initialize the first (and only, for now) G record
    psuTmatsInfo->psuFirstGRecord = (SuGRecord *)TmatsMalloc(sizeof(SuGRecord));
    psuTmatsInfo->psuFirstGRecord->szProgramName       = m_szEmpty;
    psuTmatsInfo->psuFirstGRecord->szIrig106Rev        = m_szEmpty;
    psuTmatsInfo->psuFirstGRecord->iNumDataSources     = 0;
    psuTmatsInfo->psuFirstGRecord->psuFirstGDataSource = NULL;

    // Buffer starts past Channel Specific Data
```

```
achInBuff    = (char *)pvBuff;
iInBuffIdx   = 4;

// Loop until we get to the end of the buffer
while (bTRUE)
    {

    // If at the end of the buffer then break out of the big loop
    if (iInBuffIdx >= psuHeader->ulDataLen)
        break;

    // Fill a local buffer with one line
    // --------------------------------

    // Initialize input line buffer
    szLine[0] = '\0';
    iLineIdx  = 0;

    // Read from buffer until complete line
    while (bTRUE)
        {
        // If at the end of the buffer then break out
        if (iInBuffIdx >= psuHeader->ulDataLen)
            break;

        // If line terminator and line buffer not empty then break out
        if ((achInBuff[iInBuffIdx] == CR)  ||
            (achInBuff[iInBuffIdx] == LF))
            {
            if (strlen(szLine) != 0)
                break;
            } // end if line terminator

        // Else copy next character to line buffer
        else
            {
            szLine[iLineIdx] = achInBuff[iInBuffIdx];
            if (iLineIdx < 2048)
              iLineIdx++;
            szLine[iLineIdx] = '\0';
            }

        // Next character from buffer
        iInBuffIdx++;

        } // end while filling complete line

    // Decode the TMATS line
    // ---------------------

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

    // If errors tokenizing the line then skip over them
    if ((szCodeName == NULL) || (szDataItem == NULL))
        continue;

    // Determine and decode different TMATS types
    switch (szCodeName[0])
    {
        case 'G' : // General Information
            bParseError = bDecodeGLine(szCodeName,
```

```
                                     szDataItem,
                                     &psuTmatsInfo->psuFirstGRecord);
                break;

            case 'B' : // Bus Data Attributes
                bParseError = bDecodeBLine(szCodeName,
                                     szDataItem,
                                     &psuTmatsInfo->psuFirstBRecord);
                break;

            case 'R' : // Tape/Storage Source Attributes
                bParseError = bDecodeRLine(szCodeName,
                                     szDataItem,
                                     &psuTmatsInfo->psuFirstRRecord);
                break;

            case 'T' : // Transmission Attributes
                break;

            case 'M' : // Multiplexing/Modulation Attributes
                bParseError = bDecodeMLine(szCodeName,
                                     szDataItem,
                                     &psuTmatsInfo->psuFirstMRecord);
                break;

            case 'P' : // PCM Format Attributes
                break;

            case 'D' : // PCM Measurement Description
                break;

            case 'S' : // Packet Format Attributes
                break;

            case 'A' : // PAM Attributes
                break;

            case 'C' : // Data Conversion Attributes
                break;

            case 'H' : // Airborne Hardware Attributes
                break;

            case 'V' : // Vendor Specific Attributes
                break;

            default :
                break;

            } // end decoding switch

        } // end looping forever on reading TMATS buffer

    // Now link the various records together into a tree
    vConnectRtoG(psuTmatsInfo->psuFirstGRecord, psuTmatsInfo->psuFirstRRecord);
    vConnectMtoR(psuTmatsInfo->psuFirstRRecord, psuTmatsInfo->psuFirstMRecord);
    vConnectBtoM(psuTmatsInfo->psuFirstMRecord, psuTmatsInfo->psuFirstBRecord);

    m_psuTmatsInfo = NULL;

    return I106_OK;
    }
```

```
/* -----------------------------------------------------------------------
 * G Records
 * -----------------------------------------------------------------------
 */

int bDecodeGLine(char * szCodeName, char * szDataItem, SuGRecord ** ppsuGRecord)
    {
    char          * szCodeField;
    int             iTokens;
    int             iDSIIndex;
    SuGRecord     * psuGRec;
    SuGDataSource * psuDataSource;

    // See which G field it is
    szCodeField = strtok(szCodeName, "\\");
    assert(szCodeField[0] == 'G');

    // Get the G record
    psuGRec = *ppsuGRecord;

    szCodeField = strtok(NULL, "\\");

    // PN - Program Name
    if      (strcasecmp(szCodeField, "PN") == 0)
        {
        psuGRec->szProgramName = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuGRec->szProgramName != NULL);
        strcpy(psuGRec->szProgramName, szDataItem);
        }

    // 106 - IRIG 106 rev level
    else if (strcasecmp(szCodeField, "106") == 0)
        {
        psuGRec->szIrig106Rev = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuGRec->szIrig106Rev != NULL);
        strcpy(psuGRec->szIrig106Rev, szDataItem);
        } // end if 106

    // DSI - Data source identifier info
    else if (strcasecmp(szCodeField, "DSI") == 0)
        {
        szCodeField = strtok(NULL, "\\");
        // N - Number of data sources
        if (strcasecmp(szCodeField, "N") == 0)
            psuGRec->iNumDataSources = atoi(szDataItem);
        } // end if DSI

    // DSI-n - Data source identifiers
    else if (strncasecmp(szCodeField, "DSI-",4) == 0)
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetGDataSource(psuGRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->szDataSourceID = (char *)TmatsMalloc(strlen(szDataItem)+1);
            assert(psuDataSource->szDataSourceID != NULL);
            strcpy(psuDataSource->szDataSourceID, szDataItem);
            } // end if DSI Index found
        } // end if DSI-n
```

```
    // DST-n - Data source type
    else if (strncasecmp(szCodeField, "DST-",4) == 0)
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetGDataSource(psuGRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->szDataSourceType = (char *)TmatsMalloc(strlen(szDataItem)+1);
            assert(psuDataSource->szDataSourceType != NULL);
            strcpy(psuDataSource->szDataSourceType, szDataItem);
            } // end if DSI Index found
        } // end if DST-n


    return 0;
    }

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

// Return the G record Data Source record with the given index or
// make a new one if necessary.

SuGDataSource * psuGetGDataSource(SuGRecord * psuGRecord, int iDSIIndex, int bMakeNew)
    {
    SuGDataSource   **ppsuDataSrc = &(psuGRecord->psuFirstGDataSource);

    // Walk the linked list of data sources, looking for a match or
    // the end of the list
    while (bTRUE)
        {
        // If record pointer in linked list is null then exit
        if (*ppsuDataSrc == NULL)
            {
            break;
            }

        // If the data source number matched then record found, exit
        if ((*ppsuDataSrc)->iDataSourceNum == iDSIIndex)
            {
            break;
            }

        // Not found but next record exists so make it our current pointer
        ppsuDataSrc = &((*ppsuDataSrc)->psuNextGDataSource);
        } // end

    // If no record found then put a new one on the end of the list
    if ((*ppsuDataSrc == NULL) && (bMakeNew == bTRUE))
        {
        // Allocate memory for the new record
        *ppsuDataSrc = (SuGDataSource *)TmatsMalloc(sizeof(SuGDataSource));
        assert(*ppsuDataSrc != NULL);

        // Now initialize some fields
        (*ppsuDataSrc)->iDataSourceNum     = iDSIIndex;
        (*ppsuDataSrc)->szDataSourceID     = m_szEmpty;
        (*ppsuDataSrc)->szDataSourceType   = m_szEmpty;
        (*ppsuDataSrc)->psuRRecord         = NULL;
        (*ppsuDataSrc)->psuNextGDataSource = NULL;
        }

    return *ppsuDataSrc;
```

```
    }



/* -----------------------------------------------------------------------
 * R Records
 * -----------------------------------------------------------------------
 */

int bDecodeRLine(char * szCodeName, char * szDataItem, SuRRecord ** ppsuFirstRRecord)
    {
    char          * szCodeField;
    int             iTokens;
    int             iRIdx;
    int             iDSIIndex;
    SuRRecord     * psuRRec;
    SuRDataSource * psuDataSource;

    // See which R field it is
    szCodeField = strtok(szCodeName, "\\");
    assert(szCodeField[0] == 'R');

    // Get the R record index number
    iTokens = sscanf(szCodeField, "%*1c-%i", &iRIdx);
    if (iTokens == 1)
        {
        psuRRec = psuGetRRecord(ppsuFirstRRecord, iRIdx, bTRUE);
        assert(psuRRec != NULL);
        }
    else
        return 1;

    szCodeField = strtok(NULL, "\\");

    // ID - Data source ID
    if     (strcasecmp(szCodeField, "ID") == 0)
        {
        psuRRec->szDataSourceID = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuRRec->szDataSourceID != NULL);
        strcpy(psuRRec->szDataSourceID, szDataItem);
        } // end if N

    // N - Number of data sources
    else if (strcasecmp(szCodeField, "N") == 0)
        {
        psuRRec->iNumDataSources = atoi(szDataItem);
        } // end if N

    // DSI-n - Data source identifier
    else if (strncasecmp(szCodeField, "DSI-",4) == 0)
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetRDataSource(psuRRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->szDataSourceID = (char *)TmatsMalloc(strlen(szDataItem)+1);
            assert(psuDataSource->szDataSourceID != NULL);
            strcpy(psuDataSource->szDataSourceID, szDataItem);
            } // end if DSI Index found
        else
            return 1;
        } // end if DSI-n
```

```
    // CDT-n/DST-n - Channel data type
    // A certain vendor who will remain nameless (mainly because I don't
    // know which one) encodes the channel data type as a Data Source
    // Type.  This appears to be incorrect according to the Chapter 9
    // spec but can be readily found in Chapter 10 data files.
    else if ((strncasecmp(szCodeField, "CDT-",4) == 0) ||
             (strncasecmp(szCodeField, "DST-",4) == 0))
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetRDataSource(psuRRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->szChannelDataType = (char *)TmatsMalloc(strlen(szDataItem)+1);
            assert(psuDataSource->szChannelDataType != NULL);
            strcpy(psuDataSource->szChannelDataType, szDataItem);
            } // end if DSI Index found
        else
            return 1;
        } // end if DST-n

    // TK1-n - Track number / Channel number
    else if (strncasecmp(szCodeField, "TK1-",4) == 0)
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetRDataSource(psuRRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->iTrackNumber = atoi(szDataItem);
            } // end if DSI Index found
        else
            return 1;
        } // end if TK1-n

    // CHE-n - Channel Enabled
    else if (strncasecmp(szCodeField, "CHE-",4) == 0)
        {
        iTokens = sscanf(szCodeField, "%*3c-%i", &iDSIIndex);
        if (iTokens == 1)
            {
            psuDataSource = psuGetRDataSource(psuRRec, iDSIIndex, bTRUE);
            assert(psuDataSource != NULL);
            psuDataSource->bEnabled = (strncasecmp(szDataItem, "T",1) == 0);
            } // end if DSI Index found
        else
            return 1;
        } // end if CHE-n

    return 0;
    }



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

SuRRecord * psuGetRRecord(SuRRecord ** ppsuFirstRRecord, int iRIndex, int bMakeNew)
    {
    SuRRecord   ** ppsuCurrRRec = ppsuFirstRRecord;

    // Loop looking for matching index number or end of list
    while (bTRUE)
```

```
        {
        // Check for end of list
        if (*ppsuCurrRRec == NULL)
            break;

        // Check for matching index number
        if ((*ppsuCurrRRec)->iRecordNum == iRIndex)
            break;

        // Move on to the next record in the list
        ppsuCurrRRec = &((*ppsuCurrRRec)->psuNextRRecord);
        }

    // If no record found then put a new one on the end of the list
    if ((*ppsuCurrRRec == NULL) && (bMakeNew == bTRUE))
        {
        // Allocate memory for the new record
        *ppsuCurrRRec = (SuRRecord *)TmatsMalloc(sizeof(SuRRecord));
        assert(*ppsuCurrRRec != NULL);

        // Now initialize some fields
        (*ppsuCurrRRec)->iRecordNum          = iRIndex;
        (*ppsuCurrRRec)->szDataSourceID      = m_szEmpty;
        (*ppsuCurrRRec)->iNumDataSources     = 0;
        (*ppsuCurrRRec)->psuFirstDataSource = NULL;
        (*ppsuCurrRRec)->psuNextRRecord      = NULL;
        }

    return *ppsuCurrRRec;
    }



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

// Return the R record Data Source record with the given index or
// make a new one if necessary.

SuRDataSource * psuGetRDataSource(SuRRecord * psuRRecord, int iDSIIndex, int bMakeNew)
    {
    SuRDataSource   ** ppsuDataSrc = &(psuRRecord->psuFirstDataSource);

    // Walk the linked list of data sources, looking for a match or
    // the end of the list
    while (bTRUE)
        {
        // If record pointer in linked list is null then exit
        if (*ppsuDataSrc == NULL)
            {
            break;
            }

        // If the data source number matched then record found, exit
        if ((*ppsuDataSrc)->iDataSourceNum == iDSIIndex)
            {
            break;
            }

        // Not found but next record exists so make it our current pointer
        ppsuDataSrc = &((*ppsuDataSrc)->psuNextRDataSource);
        } // end

    // If no record found then put a new one on the end of the list
```

```
    if ((*ppsuDataSrc == NULL) && (bMakeNew == bTRUE))
        {
        // Allocate memory for the new record
        *ppsuDataSrc = (SuRDataSource *)TmatsMalloc(sizeof(SuRDataSource));
        assert(*ppsuDataSrc != NULL);

        // Now initialize some fields
        (*ppsuDataSrc)->iDataSourceNum     = iDSIIndex;
        (*ppsuDataSrc)->szDataSourceID     = m_szEmpty;
        (*ppsuDataSrc)->szChannelDataType  = m_szEmpty;
        (*ppsuDataSrc)->iTrackNumber       = 0;
        (*ppsuDataSrc)->psuMRecord         = NULL;
        (*ppsuDataSrc)->psuNextRDataSource = NULL;
        }

    return *ppsuDataSrc;
    }



/* -----------------------------------------------------------------------
 * M Records
 * -----------------------------------------------------------------------
 */

int bDecodeMLine(char * szCodeName, char * szDataItem, SuMRecord ** ppsuFirstMRecord)
    {
    char          * szCodeField;
    int             iTokens;
    int             iRIdx;
    SuMRecord     * psuMRec;

    // See which M field it is
    szCodeField = strtok(szCodeName, "\\");
    assert(szCodeField[0] == 'M');

    // Get the M record index number
    iTokens = sscanf(szCodeField, "%*1c-%i", &iRIdx);
    if (iTokens == 1)
        {
        psuMRec = psuGetMRecord(ppsuFirstMRecord, iRIdx, bTRUE);
        assert(psuMRec != NULL);
        }
    else
        return 1;

    szCodeField = strtok(NULL, "\\");

    // ID - Data source ID
    if     (strcasecmp(szCodeField, "ID") == 0)
        {
        psuMRec->szDataSourceID = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuMRec->szDataSourceID != NULL);
        strcpy(psuMRec->szDataSourceID, szDataItem);
        } // end if ID

    // BSG1 - Baseband signal type
    else if (strcasecmp(szCodeField, "BSG1") == 0)
        {
        psuMRec->szBasebandSignalType = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuMRec->szBasebandSignalType != NULL);
        strcpy(psuMRec->szBasebandSignalType, szDataItem);
        } // end if BSG1
```

```
    // BB\DLN - Data link name
    else if (strncasecmp(szCodeField, "BB",2) == 0)
        {
        szCodeField = strtok(NULL, "\\");
        // DLN - Data link name
        if (strcasecmp(szCodeField, "DLN") == 0)
            {
            psuMRec->szDataLinkName = (char *)TmatsMalloc(strlen(szDataItem)+1);
            assert(psuMRec->szDataLinkName != NULL);
            strcpy(psuMRec->szDataLinkName, szDataItem);
            }
        } // end if BB\DLN

    return 0;
    }



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

SuMRecord * psuGetMRecord(SuMRecord ** ppsuFirstMRecord, int iRIndex, int bMakeNew)
    {
    SuMRecord   ** ppsuCurrMRec = ppsuFirstMRecord;

    // Loop looking for matching index number or end of list
    while (bTRUE)
        {
        // Check for end of list
        if (*ppsuCurrMRec == NULL)
            break;

        // Check for matching index number
        if ((*ppsuCurrMRec)->iRecordNum == iRIndex)
            break;

        // Move on to the next record in the list
        ppsuCurrMRec = &((*ppsuCurrMRec)->psuNextMRecord);
        }

    // If no record found then put a new one on the end of the list
    if ((*ppsuCurrMRec == NULL) && (bMakeNew == bTRUE))
        {
        // Allocate memory for the new record
        *ppsuCurrMRec = (SuMRecord *)TmatsMalloc(sizeof(SuMRecord));
        assert(*ppsuCurrMRec != NULL);

        // Now initialize some fields
        (*ppsuCurrMRec)->iRecordNum           = iRIndex;
        (*ppsuCurrMRec)->szDataSourceID       = m_szEmpty;
        (*ppsuCurrMRec)->szDataLinkName       = m_szEmpty;
        (*ppsuCurrMRec)->szBasebandSignalType = m_szEmpty;
        (*ppsuCurrMRec)->psuNextMRecord       = NULL;
        }

    return *ppsuCurrMRec;
    }



/* -----------------------------------------------------------------------
 * B Records
 * -----------------------------------------------------------------------
```

```
 */

int bDecodeBLine(char * szCodeName, char * szDataItem, SuBRecord ** ppsuFirstBRecord)
    {
    char          * szCodeField;
    int             iTokens;
    int             iRIdx;
    SuBRecord     * psuBRec;

    // See which B field it is
    szCodeField = strtok(szCodeName, "\\");
    assert(szCodeField[0] == 'B');

    // Get the B record index number
    iTokens = sscanf(szCodeField, "%*1c-%i", &iRIdx);
    if (iTokens == 1)
        {
        psuBRec = psuGetBRecord(ppsuFirstBRecord, iRIdx, bTRUE);
        assert(psuBRec != NULL);
        }
    else
        return 1;

    szCodeField = strtok(NULL, "\\");

    // DLN - Data link name
    if     (strcasecmp(szCodeField, "DLN") == 0)
        {
        psuBRec->szDataLinkName = (char *)TmatsMalloc(strlen(szDataItem)+1);
        assert(psuBRec->szDataLinkName != NULL);
        strcpy(psuBRec->szDataLinkName, szDataItem);
        } // end if DLN

    // NBS\N - Number of buses
    else if (strncasecmp(szCodeField, "NBS",3) == 0)
        {
        szCodeField = strtok(NULL, "\\");
        // N - Number of channels
        if (strcasecmp(szCodeField, "N") == 0)
            {
            psuBRec->iNumBuses = atoi(szDataItem);
            }
        } // end if NBS

    return 0;
    }



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

SuBRecord * psuGetBRecord(SuBRecord ** ppsuFirstBRecord, int iRIndex, int bMakeNew)
    {
    SuBRecord   ** ppsuCurrBRec = ppsuFirstBRecord;

    // Loop looking for matching index number or end of list
    while (bTRUE)
        {
        // Check for end of list
        if (*ppsuCurrBRec == NULL)
            break;

        // Check for matching index number
```

```
        if ((*ppsuCurrBRec)->iRecordNum == iRIndex)
            break;

        // Move on to the next record in the list
        ppsuCurrBRec = &((*ppsuCurrBRec)->psuNextBRecord);
        }

    // If no record found then put a new one on the end of the list
    if ((*ppsuCurrBRec == NULL) && (bMakeNew == bTRUE))
        {
        // Allocate memory for the new record
        *ppsuCurrBRec = (SuBRecord *)TmatsMalloc(sizeof(SuBRecord));
        assert(*ppsuCurrBRec != NULL);

        // Now initialize some fields
        (*ppsuCurrBRec)->iRecordNum          = iRIndex;
        (*ppsuCurrBRec)->szDataLinkName      = m_szEmpty;
        (*ppsuCurrBRec)->iNumBuses           = 0;
        (*ppsuCurrBRec)->psuNextBRecord      = NULL;
        }

    return *ppsuCurrBRec;
    }



/* -----------------------------------------------------------------------
 * Connect records into a tree structure
 * -----------------------------------------------------------------------
 */

// Connect R records with the coresponding G data source record.

void vConnectRtoG(SuGRecord * psuFirstGRecord, SuRRecord * psuFirstRRecord)
    {
    SuRRecord       * psuCurrRRec;
    SuGDataSource   * psuCurrGDataSrc;

    // Walk through the R record linked list, looking for a match to the
    // appropriate G data source record.
    psuCurrRRec = psuFirstRRecord;
    while (psuCurrRRec != NULL)
        {

        // Step through the G data source records looking for a match
        psuCurrGDataSrc = psuFirstGRecord->psuFirstGDataSource;
        while (psuCurrGDataSrc != NULL)
            {
            // See if IDs match
            if (strcasecmp(psuCurrGDataSrc->szDataSourceID,
                         psuCurrRRec->szDataSourceID) == 0)
                {
// If psuCurrGDataSrc->psuRRecord != NULL then that is probably an error in the TMATS file
                psuCurrGDataSrc->psuRRecord = psuCurrRRec;
// If R can't connect to more than one G then we could break here.
                } // end if match

            // Get the next G data source record
            psuCurrGDataSrc = psuCurrGDataSrc->psuNextGDataSource;
            } // end while walking the G data source records

        // Get the next R record
        psuCurrRRec = psuCurrRRec->psuNextRRecord;
```

```
        } // end while walking the R record list

    return;
    }



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

void vConnectMtoR(SuRRecord * psuFirstRRecord, SuMRecord * psuFirstMRecord)
    {
    SuMRecord       * psuCurrMRec;
    SuRRecord       * psuCurrRRec;
    SuRDataSource   * psuCurrRDataSrc;

    // Walk through the M record linked list, looking for a match to the
    // appropriate R data source record.
    psuCurrMRec = psuFirstMRecord;
    while (psuCurrMRec != NULL)
        {

        // Walk the linked list of R records
        psuCurrRRec = psuFirstRRecord;
        while (psuCurrRRec != NULL)
            {

            // Walk the linked list of R data sources
            psuCurrRDataSrc = psuCurrRRec->psuFirstDataSource;
            while (psuCurrRDataSrc != NULL)
                {

                // See if IDs match
                if (strcasecmp(psuCurrRDataSrc->szDataSourceID,
                               psuCurrMRec->szDataLinkName) == 0)
                    {
// If psuCurrRDataSrc->psuMRecord != NULL then that is probably an error in the TMATS file
                    psuCurrRDataSrc->psuMRecord = psuCurrMRec;
// If M can't connect to more than one R then we could break here.
                    } // end if match

                // Get the next R data source record
                psuCurrRDataSrc = psuCurrRDataSrc->psuNextRDataSource;
                } // end while walking the R data source records

            // Get the next R record
            psuCurrRRec = psuCurrRRec->psuNextRRecord;
            }

        // Get the next M record
        psuCurrMRec = psuCurrMRec->psuNextMRecord;
        } // end while walking the M record list

    return;
    }



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

void vConnectBtoM(SuMRecord * psuFirstMRecord, SuBRecord * psuFirstBRecord)
    {
    SuBRecord       * psuCurrBRec;
```

```
    SuMRecord       * psuCurrMRec;

    // Walk through the B record linked list, looking for a match to the
    // appropriate M data source record.
    psuCurrBRec = psuFirstBRecord;
    while (psuCurrBRec != NULL)
        {

        // Walk the linked list of M records
        psuCurrMRec = psuFirstMRecord;
        while (psuCurrMRec != NULL)
            {

            // See if IDs match
            if (strcasecmp(psuCurrMRec->szDataLinkName,
                           psuCurrBRec->szDataLinkName) == 0)
                {
// If psuCurrMRecord->psuBRecord != NULL then that is probably an error in the TMATS file
                psuCurrMRec->psuBRecord = psuCurrBRec;
// If B can't connect to more than one M then we could break here.
                } // end if match

            // Get the next R record
            psuCurrMRec = psuCurrMRec->psuNextMRecord;
            }

        // Get the next M record
        psuCurrBRec = psuCurrBRec->psuNextBRecord;
        } // end while walking the M record list

    return;
    }

// -----------------------------------------------------------------------

// The enI106_Decode_Tmats() procedure malloc()'s a lot of memory.  This
// procedure walks the SuMemBlock list, freeing memory as it goes.

void I106_CALL_DECL
    enI106_Free_TmatsInfo(SuTmatsInfo    * psuTmatsInfo)
    {
    SuMemBlock     * psuCurrMemBlock;
    SuMemBlock     * psuNextMemBlock;

    if (psuTmatsInfo == NULL)
        return;

    // Walk the linked memory list, freely freeing as we head down the freeway
    psuCurrMemBlock = psuTmatsInfo->psuFirstMemBlock;
    while (psuCurrMemBlock != NULL)
        {
        // Free the memory
        free(psuCurrMemBlock->pvMemBlock);

        // Free the memory block and move to the next one
        psuNextMemBlock = psuCurrMemBlock->psuNextMemBlock;
        free(psuCurrMemBlock);
        psuCurrMemBlock = psuNextMemBlock;
        }

    // Initialize the TMATS info data structure
    memset(psuTmatsInfo, 0, sizeof(SuTmatsInfo));
```

```
    return;
    }



// -----------------------------------------------------------------------

// Allocate memory but keep track of it for enI106_Free_TmatsInfo() later.

void * TmatsMalloc(size_t iSize)
    {
    void             * pvNewBuff;
    SuMemBlock      ** ppsuCurrMemBlock;

    // Malloc the new memory
    pvNewBuff = malloc(iSize);

    // Walk to (and point to) the last linked memory block
    ppsuCurrMemBlock = &m_psuTmatsInfo->psuFirstMemBlock;
    while (*ppsuCurrMemBlock != NULL)
        ppsuCurrMemBlock = &(*ppsuCurrMemBlock)->psuNextMemBlock;

    // Populate the memory block struct
    *ppsuCurrMemBlock = (SuMemBlock *)malloc(sizeof(SuMemBlock));
    (*ppsuCurrMemBlock)->pvMemBlock      = pvNewBuff;
    (*ppsuCurrMemBlock)->psuNextMemBlock = NULL;

    return pvNewBuff;
    }



/* -----------------------------------------------------------------------
 * Write procedures
 * ----------------------------------------------------------------------- */

I106_CALL_DECL EnI106Status
    enI106_Encode_Tmats(SuI106Ch10Header * psuHeader,
                        void             * pvBuff,
                        char             * szTMATS)
    {
    // Channel specific data word
    *(uint32_t *)pvBuff = 0;

    // Figure out the total TMATS message length
    psuHeader->ulDataLen = strlen(szTMATS) + 4;

    // Copy TMATS setup info to buffer.  This assumes there is enough
    // space in the buffer to hold the TMATS string.
    strcpy((char *)pvBuff+4, szTMATS);

    // Make the data buffer checksum and update the header
    uAddDataFillerChecksum(psuHeader, (unsigned char *)pvBuff);

    return I106_OK;
    }
```

This page intentionally left blank.

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

// .NET 2005 C++ wants structures that are passed as function parameters to be declared
// as public.  .NET 2003 and native C pukes on that. C++ Interop doesn't seem to care.
//  Grrrr...  Just define out PUBLIC for now but leave in the macro logic in case I want
// to revisit this someday.  Yeah, right!
//#if _MSC_VER >= 1400
//#define PUBLIC public
//#else
#define PUBLIC
//#endif

// .NET 2005 (and probably earlier, but I'm not sure) define time_t to be a 64 bit value.
// And by default, all the CRT time routines are the 64 bit versions.  For best portability,
// time_t is assumed to be a 32 bit value.  The following #define tells .NET to use 32 bits
// as the default time_t size.  This needs to be set in the project properties.  This forces
// a puke if it isn't set.
#if _MSC_VER >= 1400
  #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

#ifdef __cplusplus
}
#endif

#endif
```

APPENDIX A-10 - STDINT.H

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

 stdint.h - Define standard size integers

 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 _user_stdint_h
#define _user_stdint_h

// Modern versions of GCC usually have stdint.h so include it instead
#if defined(__GNUC__) && !defined(__DJGPP__)
#include <stdint.h>
#endif

// The DJGPP
#if defined(__DJGPP__)
typedef char                int8_t;
typedef short               int16_t;
typedef int                 int32_t;
typedef long long           int64_t;

typedef unsigned char       uint8_t;
typedef unsigned short      uint16_t;
typedef unsigned int        uint32_t;
typedef unsigned long long  uint64_t;
#endif

// Define specific sized variables for MSVC
#if defined(_WIN32)
typedef __int8              int8_t;
```

```
typedef __int16             int16_t;
typedef __int32             int32_t;
typedef __int64             int64_t;

typedef unsigned __int8     uint8_t;
typedef unsigned __int16    uint16_t;
typedef unsigned __int32    uint32_t;
typedef unsigned __int64    uint64_t;
#endif

#endif
```

APPENDIX B

EXAMPLE PROGRAM - CALCULATE HISTOGRAM

The software program, shown on the following pages, 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.

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

#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_tmats.h"


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

#define MAJOR_VERSION  "B1"
#define MINOR_VERSION  "02"

#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       ulTotalIrigMsgs;
    unsigned long       ulTotalBusMsgs;
    unsigned long       aulMsgs[0x4000];
    unsigned long       aulErrs[0x4000];
    unsigned long       ulErr1553Timeout;
    int                 bRT2RTFound;
    } SuChanInfo1553;

// 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       ulOther;
    } SuChanInfo;


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

int                 m_bLogRT2RT;
int                 m_bVerbose;

/*
 * Function prototypes
 * -------------------
 */

void     vPrintCounts(SuChanInfo * psuChanInfo, FILE * ptOutFile);
void     vPrintTmats(SuTmatsInfo * psuTmatsInfo, FILE * ptOutFile);
void     vProcessTmats(SuTmatsInfo * psuTmatsInfo, SuChanInfo * apsuChanInfo[]);
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           ulTotal;

    FILE                  * ptOutFile;        // Output file handle
    int                     hI106In;
    char                    szInFile[80];     // Input file name
    char                    szOutFile[80];    // 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;
    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;

/*
 * 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)
        {
        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;
        }


    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))
        {
        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
        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 :               // 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->ulTotalIrigMsgs++;

    // Step through all 1553 messages
    enStatus = enI106_Decode_First1553F1(&suI106Hdr, pvBuff, &su1553Msg);
    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++;

                        // Get the next 1553 message
                        enStatus = enI106_Decode_Next1553F1(&su1553Msg);
                        }

                        // 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

                    break;

                case I106CH10_DTYPE_ANALOG :            // 0x21
                    apsuChanInfo[suI106Hdr.uChID]->ulAnalog++;
                    break;

                case I106CH10_DTYPE_VIDEO_FMT_0 :       // 0x40
                    apsuChanInfo[suI106Hdr.uChID]->ulMPEG2++;
                    break;

                case I106CH10_DTYPE_UART_FMT_0 :        // 0x50
                    apsuChanInfo[suI106Hdr.uChID]->ulUART++;
                    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, ptOutFile);

    fprintf(ptOutFile,"\n=-=-= Message Totals by Channel and Type =-=-=\n\n");
    for (uChanIdx=0; uChanIdx<0x1000; uChanIdx++)
        {
        if (apsuChanInfo[uChanIdx] != NULL)
            {
```

```
            vPrintCounts(apsuChanInfo[uChanIdx], ptOutFile);
            }
        }


    fprintf(ptOutFile,"=-=-= 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(ptOutFile,"File Start %s\n",  szTime);

    enI106_Rel2IrigTime(hI106In, abyStartTime, &suIrigTime);
    psuTmTime = gmtime((time_t *)&(suIrigTime.ulSecs));
    strftime(szTime, 50, szTimeFmt, psuTmTime);
    fprintf(ptOutFile,"Data Start %s\n",  szTime);

    enI106_Rel2IrigTime(hI106In, abyStopTime, &suIrigTime);
    psuTmTime = gmtime((time_t *)&(suIrigTime.ulSecs));
    strftime(szTime, 50, szTimeFmt, psuTmTime);
    fprintf(ptOutFile,"Data Stop  %s\n\n",  szTime);

    fprintf(ptOutFile,"\nTOTAL RECORDS:    %10lu\n\n", ulTotal);

/*
 *  Free dynamic memory.
 */

    free(pvBuff);
    pvBuff = NULL;

    fclose(ptOutFile);

    return 0;
    }



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

void vPrintCounts(SuChanInfo * psuChanInfo, FILE * ptOutFile)
    {
    long            lMsgIdx;

    // Make Channel ID line lead-in string
    fprintf(ptOutFile,"ChanID %3d : %s : %s\n",
        psuChanInfo->iChanID, psuChanInfo->szChanType, psuChanInfo->szChanName);

    if (psuChanInfo->ulTMATS != 0)
        fprintf(ptOutFile,"    TMATS             %10lu\n",   psuChanInfo->ulTMATS);

    if (psuChanInfo->ulEvents != 0)
        fprintf(ptOutFile,"    Events            %10lu\n",   psuChanInfo->ulEvents);

    if (psuChanInfo->ulIndex != 0)
        fprintf(ptOutFile,"    Index             %10lu\n",   psuChanInfo->ulIndex);

    if (psuChanInfo->ulIrigTime != 0)
        fprintf(ptOutFile,"    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(ptOutFile,"    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

        if (psuChanInfo->psu1553Info->bRT2RTFound == bTRUE)
            {
            fprintf(ptOutFile,"\n  Warning - RT to RT transfers found in the data\n");
            if (m_bLogRT2RT == bTRUE)
                fprintf(ptOutFile,
                        "    Message total is NOT the sum of individual RT totals\n");
            else
                fprintf(ptOutFile,
                        "    Some transmit RTs may not be shown\n");
            } // end if RT to RT

        fprintf(ptOutFile,"    Totals - %ld Message in %ld IRIG Records\n",
            psuChanInfo->psu1553Info->ulTotalBusMsgs,
            psuChanInfo->psu1553Info->ulTotalIrigMsgs);
        } // end if 1553 messages

    if (psuChanInfo->ulPCM != 0)
        fprintf(ptOutFile,"    PCM              %10lu\n",   psuChanInfo->ulPCM);

    if (psuChanInfo->ulAnalog != 0)
        fprintf(ptOutFile,"    Analog           %10lu\n",   psuChanInfo->ulAnalog);

    if (psuChanInfo->ulMPEG2 != 0)
        fprintf(ptOutFile,"    MPEG Video       %10lu\n",   psuChanInfo->ulMPEG2);

    if (psuChanInfo->ulUART != 0)
        fprintf(ptOutFile,"    UART             %10lu\n",   psuChanInfo->ulUART);

    if (psuChanInfo->ulUserDefined != 0)
        fprintf(ptOutFile,"    User Defined     %10lu\n",   psuChanInfo->ulUserDefined);

    if (psuChanInfo->ulOther != 0)
        fprintf(ptOutFile,"    Other messages   %10lu\n",   psuChanInfo->ulOther);

    fprintf(ptOutFile,"\n",   psuChanInfo->ulOther);
    return;
    }



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

void vPrintTmats(SuTmatsInfo * psuTmatsInfo, FILE * ptOutFile)
```

```
    {
    int                     iGIndex;
    int                     iRIndex;
    int                     iRDsiIndex;
    SuGDataSource         * psuGDataSource;
    SuRRecord             * psuRRecord;
    SuRDataSource         * psuRDataSource;

    // Print out the TMATS info
    // -----------------------

    fprintf(ptOutFile,"\n=-=-= Channel Summary =-=-=\n\n");

    // G record
    fprintf(ptOutFile,"Program Name - %s\n",psuTmatsInfo->psuFirstGRecord->szProgramName);
    fprintf(ptOutFile,"IRIG 106 Rev - %s\n",psuTmatsInfo->psuFirstGRecord->szIrig106Rev);
    fprintf(ptOutFile,"Channel  Type          Data Source         \n");
    fprintf(ptOutFile,"-------  ------------  --------------------\n");

    // Data sources
    psuGDataSource = psuTmatsInfo->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,"  %-20s", psuRDataSource->szDataSourceID);
                fprintf(ptOutFile,"\n");
                psuRDataSource = psuRDataSource->psuNextRDataSource;
                } while (bTRUE);

            psuRRecord = psuRRecord->psuNextRRecord;
            } while (bTRUE);


        psuGDataSource =
            psuTmatsInfo->psuFirstGRecord->psuFirstGDataSource->psuNextGDataSource;
        } while (bTRUE);

    return;
    }



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

void vProcessTmats(SuTmatsInfo * psuTmatsInfo, SuChanInfo * apsuChanInfo[])
    {
//    unsigned            uArrayIdx;
```

```
//    unsigned char       u1553ChanIdx;
    SuRRecord         * psuRRecord;
    SuRDataSource     * psuRDataSrc;

    // 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)
            {
            // Make sure a message count structure exists
            if (apsuChanInfo[psuRDataSrc->iTrackNumber] == NULL)
                {
// SOMEDAY PROBABLY WANT TO HAVE DIFFERENT COUNT STRUCTURES FOR EACH CHANNEL TYPE
                apsuChanInfo[psuRDataSrc->iTrackNumber] = malloc(sizeof(SuChanInfo));
                memset(apsuChanInfo[psuRDataSrc->iTrackNumber], 0, sizeof(SuChanInfo));
                apsuChanInfo[psuRDataSrc->iTrackNumber]->iChanID = psuRDataSrc->iTrackNumber;
                }

            // Now save channel type and name
            strcpy(apsuChanInfo[psuRDataSrc->iTrackNumber]->szChanType,
                   psuRDataSrc->szChannelDataType);
            strcpy(apsuChanInfo[psuRDataSrc->iTrackNumber]->szChanName,
                   psuRDataSrc->szDataSourceID);

            // Get the next R record data source
            psuRDataSrc = psuRDataSrc->psuNextRDataSource;
            } // end while walking R data source linked list

        // Get the next R record
        psuRRecord = psuRRecord->psuNextRRecord;

        } // end while walking R record linked list

    return;
    }



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

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");
    }
```

This page intentionally left blank.

APPENDIX C

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 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;
    }
```