Engineering Committee Report

SMPTE Header/Descriptor Task Force: Final Report

January 3, 1992

This report of the SMPTE Task Force on Headers/Descriptors is an approved document of the SMPTE Standards Committee and is made available for information, as it contains valuable proposals concerning the development of digital imaging and video systems and for standardization of certain of their aspects that will be of interest generally. The standardization aspects of the report will be further considered under the normal processes of the SMPTE for the creation and approval of engineering documents, which includes the opportunity for further comment and for public review prior to their final acceptance. Persons wishing to actively participate in the development of these standards, including attendance at Working Group meetings and ballot response, may contact the Engineering Dept. of the SMPTE. It should be noted that engineering documents arising from the contents of this report may differ significantly from its recommendations, and caution is suggested in the use of this report as the basis of design or implementation.

1.0 Introduction

The Task Force on Header/Descriptors has considered the questions posed in its scope of committee work and makes the following final report and recommendations to the Standards Committee. The report begins with a discussion of the general objectives of the header/descriptor, and then presents more specific objectives selected by the Task Force as it developed two alternative implementations.

Both of the proposed implementations could support new SMPTE standards, and are described in some detail here. The "ASN.1 Implementation" is structured using only Abstract Syntax Notation 1 (ASN.1), an existing and evolving ISO/CCITT standard principally used in the computer industry. The "Compact Implementation" is designed to minimize the number of bits allocated to the header/descriptor function, but also permits optional use of the ASN.1 notation later in the header/descriptor for further extensibility. Both implementations perform essentially identical functions.

Appendices A and B present illustrative approaches to the design of transport headers and header-decoding software, respectively. Transport headers are designed to address certain difficult data transport problems. Appendix C lists the official task force members as of January 3, 1992.

In view of (1) the great importance to industry and its customers of the capabilities provided by the header/descriptors described below, and (2) the degree to which these two possible implementations satisfy the objectives established at the outset for header/descriptors, the Task Force recommends that: the Standards Committee arrange for the preparation of one or two new standards for digital header/descriptors based on either the "Compact" or the "ASN.1" Implementations described below, or on a combination thereof.

2.0 General Objectives

The header/descriptor task force was directed to consider header/descriptor architectures and implementations appropriate for the emerging digital high-definition television (HDTV) and high-resolution system (HRS) industries. The primary design objectives of the task force are:

- *Universality.* All image and other data streams should be labeled so that signals can be shared across systems and applications with minimal degradation or confusion; the header/descriptor should therefore uniquely identify the encoding scheme employed and how the data is to be interpreted.
- *Longevity.* The header/descriptor should provide a number of potential identification codes adequate to serve for decades, and preferably centuries; this implies that specific encoding identifiers, once assigned and registered, should not be reassigned or redefined. The header/descriptor should also facilitate longevity for equipment and media of all types.
- *Extensibility.* To facilitate service enhancement and innovation, and to promote longevity of both equipment and recorded signals, the header/descriptor should accommodate technological advances in either equipment or recorded signals with minimal risk of obsoleting existing components, infrastructure, and media collections.
- *Interoperability.* The header should permit optimal sharing of data streams across data-generation, carrier, and equipment technologies and services in a variety of error environments and should permit all equipment and applications to successfully ignore encrypted or otherwise deliberately inaccessible data.
- *Cost/performance effectiveness.* The header/descriptor should permit use of both low-cost equipment as well as more expensive high-performance equipment; the header/descriptor should also accommodate inexpensive equipment incapable of decoding all possible data streams. Economy and simplicity through flexibility and scalability of the key performance parameters should also be supportable.
- *Compactness.* The header/descriptor should be economical in its utilization of bits and should typically comprise a negligible fraction of the underlying data stream.
- *Rapid capture.* Much video and other serial data is intercepted midstream, such as when users switch to a new channel, and therefore the header/descriptor should permit rapid header identification, adequate to meet the needs of all applications.
- *Editability.* Common editing and parsing operations, such as splicing, appending, replacing, inserting, cropping, and overlays, should be supportable by the header/descriptor architecture without necessarily requiring decoding and encoding of the data stream itself.

3.0 Specific Header/Descriptor Objectives

To meet the general objectives summarized above, the Task Force selected the following compact set of specific objectives which are met by both implementations described later. The header and descriptor are defined here separately.

3.1 Specific Header Objectives

The specific objectives of the header are to:

• Identify by number the encoding standard employed by the attached block of data.

• Specify the length of that block of data, so that equipment of any epoch can successfully skip uninteresting blocks of data or data encoded using standards defined subsequently.

• Indicate whether a readable descriptor follows the header.

• Permit users to intercept data streams at random times, as when switching channels, so that proper data interpretation begins swiftly.

• Provide optional error-protection capability. Data generation entities may wish to supplement error-protection services provided in subsequent environments experienced by that data, particularly when those environments are unknown.

The task force considers these attributes of the header to be the minimum mandatory set, recognizing that additional important capabilities can be provided by the descriptor.

3.2 Examples of Header Use

A simple example illustrates how these minimal capabilities for the header satisfy the general objectives discussed above. Suppose, after many years, some HDTV broadcasters wish to provide dual-language sound tracks. This capability could be provided by adding to the data stream blocks of data conveying the second language. These new blocks would be labeled by a header incorporating a standard number not recognized by equipment produced earlier. This older equipment would read the header and recognize the standard identification number as being unknown. It could then observe the length of the associated block, and skip over it to the next header.

All data could be labeled by such flexible headers, or only a designated portion (e.g., "auxiliary data") of a more rigidly defined larger video data stream. Note that HDTV receivers capable of receiving only 20 Mbits/sec could not accommodate increases except at the expense of any spare capacity previously reserved for expansion, or by the broadcaster reducing the number of bits conveying video or audio; in the latter case the original standard would have to be defined so as to permit receivers to accommodate any such real-time video or audio truncation, however.

3.3 Specific Descriptor Objectives

The principal function of the descriptor is to convey additional information that improves the usefulness of the data to the user; its format would be specified independently of the standard employed for the data itself. Such optional auxiliary information in the descriptor might include transport information such as cryptographic, priority, or additional error-protection information, as well as source time, authorship, ownership, restrictions on use, royalty payment information, explicit description of encoding or decoding processes, intermediate processing performed, and other information in forms that could evolve over the years. To simplify the decoding task, the descriptor may also contain an abbreviated table of contents and a flag indicating whether any information has changed since the previous descriptor. The beginning of the descriptor would also indicate the descriptor length so that it might be skipped without interpretation if the user chooses. Optional additional error protection would be available for data originators so desiring it.

Specifically, the descriptor could include:

• A list of standard-identification numbers, parameters of operation, text, and algorithms, in any desired combination.

• A compact optional table of contents for the descriptor.

• A flag indicating whether changes occurred since the previous descriptor.

• The length of the descriptor so that it might be readily skipped if desired.

• Information indicating the number of descriptor entries and their formats so that they might be properly interpreted.

• Optional error protection for the descriptor.

The presence or absence of a descriptor could be indicated by one of the bits contained in the header.

3.4 Examples of Descriptor Use

The use of standard identification numbers in the descriptor permits very compact and flexible encoding. For example, one such number might be allocated internationally to each model number of studio television camera, so that subsequent image processing can maximally improve image quality, compensating for any camera idiosyncrasies. Similar identifiers could be used for different forms of physical, analog, or digital filtering that has been applied to the image subsequently, so that user equipment might again appropriately refilter the image in an optimum way depending on the user's intentions. This is important because performance for any particular display device or audio system is best when it reflects the processing that has occurred previously.

Authorship, ownership, and other such information could be conveyed by compact standard-identification numbers, or by use of plain text in English or another language. Certain descriptors might simply be numbers indicating the settings of certain switches at the time the signals were generated, such as switches controlling audio bass, treble, or volume. The descriptor may also include subroutines or other encoded instructions that facilitate subsequent processing or decoding.

Standards numbers and parameter fields can also be used to support transport-layer functions, including essentially all forms of cryptography, statement of the relative priority of the current data block, priority "bidding" data (so users can bid for priority in a free-market sense), synchronization reinforcement blocks, and other information, the character of which can be defined over the years as new standards identification numbers are assigned and as new languages and protocols are defined. The incorporation of transport capabilities in this standard should not compromise those established by other layers, but would merely supplement them. The only limitation is that such descriptor transport standards should remain robust if block sequences are shuffled in another transport layer.

4.0 Illustrative Examples of Header/Descriptor Use

An HDTV broadcaster could simply divide the HDTV signal into blocks, each beginning with a header of perhaps 6 to 16 bytes in length. This header would contain the length of each block, which could be fixed for all time or variable, and a unique standards number indicating the given HDTV encoding protocol, which may also be unchanging in the initial years. If the over-the-air broadcast standard is heavily error protected, little additional error protection might be added to the header. Good engineering practice would suggest, however, that the header be independently error protected using some of the options described later, and that separate, and possibly less robust, error protection be applied to the remainder of the data stream.

If the HDTV channel is defined so as to perform all transport functions, including all synchronization and error correction, then the header/descriptor described here might be embedded in the transported data stream. At that level it would preferably be used to encapsulate all data, but could be used in an inferior implementation to encapsulate and characterize only substream or side-channel data. In this example too, the descriptor might convey the origins and processing history of the data, enabling future higher-performance systems to employ post-filters optimizing the quality of the output images. Such flexibility could be particularly important if the output display capabilities of the equipment enabled flexibility in frame rate, pixel interpolation routines, and chrominance manipulation. In the case of audio signals, descriptors could be used in a similar manner to enable optimum reproduction by characterizing microphone placement and preprocessing.

A variety of header/descriptors plus associated data could be sequenced in any side channel to provide a variety of flexible delivered products, such as multiple audio channels, multiple-language captioning,

home-shopping order information, and even entire TV signals encoded at lower quality so that more than one can share a single channel. In this example, the ultimate flexibility that could be obtained with individual broadcast channels would be heavily dependent upon the flexibility inherent in the broadcaster's ability to decrease the data rate associated with the initial broadcast product to accommodate possible growth in use of the side-channel capability.

It is also possible to send descriptor information in a separate block with its own header, rather than embedding it in the same blocks as the associated data. In this case the characterization of a signal provided by a descriptor could remain unchanged for many blocks until that descriptor information changes. In the event the transport layer is prone to shuffling the sequence of blocks, impairing association of descriptors and use of this data, the inherent flexibility of this header/descriptor system permits incorporation of sequence numbers in blocks so that the receiving entity can properly sequence them. Such separation of descriptor information into separate blocks also simplifies translation between environments having different transport layer protocols; any descriptor elements providing transport-layer functionality can be added or deleted when moving between such environments, as desired. An option whereby special "transport header/descriptors" are prepended to blocks is described in Section 5.2.7.

To the extent that transport-layer functionality might be embedded within the basic data block, it could be harmlessly overlaid by similar transport-layer functionality in other system elements without loss. Thus the great flexibility inherent in the header/descriptor architecture proposed here, including its ability to perform multiple functions in a multilayer ISO environment, should not be a handicap, and may in some applications be an important advantage.

Furthermore, this header/descriptor structure permits efficient utilization of the standards activities of a wide variety of national and international organizations. Every standard developed by such bodies for characterizing or communicating digital information can be characterized by an identification code that can be conveyed in an efficient manner by the header/descriptor system described here. Since the implementations proposed here call for each standards authority to have its own identification number, such standards bodies could, in this "ID" concept, choose to use the very same identification numbers they have previously chosen for other purposes. Alternatively, that authority could choose some other simple one-to-one mapping between numbers. At the same time, such standards can also use this header/descriptor system as an embedded construct within their own protocols. Although the full power of the flexible extensible structure proposed here will only become apparent as it is developed and improved over the years, the basic architecture can be fixed immediately. This would permit immediate fabrication and utilization of equipment based upon this standard.

5.0 Compact Header/Descriptor Approach

5.1 Compact Header/Descriptor Architecture

The header is divided into two parts: a 2-byte "header key," and the remainder, or "header tail." Among the available header options are those without tails, corresponding to a 2-byte header conveying simple messages; 32 possible messages of this type are available for future definition. The most usual function of the key, however, is merely to "unlock" the tail by providing information about the format of that tail.

The descriptor is divided into three parts: a 2-byte "descriptor key" (similar to the header key), a "core" of 0 to 8 bytes, and the "descriptor tail," which conveys a series of numbers signifying various pieces of information.

5.2 Header Key

5.2.1 Header Key Organization

The header key consists of two 4-bit fields and one 8-bit field. The 8-bit field provides 2-bit error-correction capability for the key, and the first 4-bit length-type "LT" field determines the length of the header field in the tail that contains the length of the block. A block of data is defined as comprising the header, any descriptors, and the associated data. The other 4-bit "ID" field normally determines the number of bits in the header-tail field devoted to indicating the standard number under which the remainder of the data in the block, possibly including the descriptor, is encoded. Most descriptors would be publicly readable, however. The ID field also indicates whether a readable descriptor follows the header; sometimes one might wish to read the descriptor before deciding whether to decode the rest of the data block. The ID field can alternatively convey messages for certain 4-bit combinations in the first 4-bit LT field. In addition to specifying the length of the header tail field devoted to specifying block length, the LT field also determines what level of error protection is being provided for the header.

5.2.2 Header Key: 4-Bit "Length-Type" LT Field

The primary purpose of the 4-bit LT field is to specify the length of the field in the header tail devoted to specifying the block length. For 6 of the 16 possible combinations proposed here (the "fixed-length" options) the total length of the block is prespecified so that no bits in the header tail are allocated to specifying block length. The other 10 proposed combinations permit use of 1, 2, 4, and 6 bytes for specifying the block length in bytes in integer format; in each case versions with and without error-protection capability for the header are available to the standards definition community. The 8-bit protection provided to the header key is always present, however, since the integrity of the key is crucial, and it represents such a small part of the total.

The six proposed fixed-length options are as follows:

1. The block is 2 bytes long and the message is conveyed by the 4-bit ID field; 16 messages are possible. One of these messages could signify that the rest of the header/descriptor is coded in ASN.1 (dual compact- and ASN.1-header decoding capability would be necessary for all equipment, however, but Appendix B suggests this might not be burdensome).

2. Same as (1), except that an additional 16 messages are possible (for a total of 32 2-byte options).

3. The block is 4 bytes long, the message consisting of the 4 ID bits plus 2 bytes, a total of 20 bits.

4. The block length is 6 bytes; 28 bits $(4+3\times8)$ are available, the remaining byte providing error protection for the last 4-byte set.

5. The block length is 6 bytes with 36 bits of information $(4+4\times8)$ being available, but without additional error protection.

6. The length of the block is unknown or irrelevant.

In each of the foregoing options, except the first two, the available bits can be divided in a yet-to-be-determined way between those indicating the standard identification number and any additional message. The proposed SMPTE standard would constrain only options 4 and 5, allocating the first 8 bits to designating the sovereign state, so that development of these 6-byte options can proceed without international agreement. Designation of standards bodies and sovereign states is discussed further in Section 5.2.4.3.

The ten remaining proposed options for the LT field provide for either 1, 2, 4, or 6 bytes in the header tail to be allocated to specifying the block length in bytes, always in integer format. Depending on which of these ten options is chosen, additional information is also conveyed concerning the level of error protection for the header. These proposed options are as follows:

7. One-byte field in the header tail specifies block length in integer format; no additional error-correction capability is provided to the header. Blocks up to 256

bytes long are available under this option.

8. Same as (7), except an additional 1 byte is provided for header error protection.

9. Same as (7), except that 2 bytes specify block length; the maximum block length here is 64 Kbytes.

10. Same as (9), with an additional byte for header error protection.

11. Same as (7), except that 4 bytes specify block length, which can approach 4 billion bytes.

12. Same as (11), except that 2 bytes of header error protection are added.

13. Same as (7), except that 6 bytes specify block length.

14. Same as (13), except that 2 bytes of header error protection are added.

15–16. To be determined.

5.2.3 Header Key: 4-Bit ID Field

When the 4-bit ID field is not being used to convey a message using LT options 1 to 6, then the first 3 bits of the ID field indicate one of eight possible lengths for the header-tail field conveying the standard identification number, and the fourth ID bit indicates whether a readable descriptor follows the header. These proposed eight tail-length field options include 1, 2, 4, and 8-byte options. These eight tail-length options proposed for the ID field are as follows:

1. One-byte standard identification number allocated internationally.

2. Same as (1), but providing for 256 additional possible standards.

3. Similar to (1), except that the standard field in the header tail contains 2 bytes instead of 1, providing for 64,000 international standards.

4. Similar to (1), but with a 3-byte field in the header tail, providing for over 16 million international standards.

5. Two bytes in the header tail indicate the standard identification number, the first byte indicating the sovereign state (see Section 5.2.4.3).

6. Similar to (5), except that 4 bytes are available (one for the sovereign state).

7. Similar to (5), except that 8 bytes are available.

8. To be determined, or reserved for the distant future.

These first four options provide 1, 2, and 3-byte standard identification numbers to some designated international standards body or bodies. Two versions of the 1-byte option are provided because otherwise too few combinations would be available. The remaining options provide 1 byte (or more) that identifies the sovereign state under whose authority the remaining standard-identification-number bytes the ID field have been assigned (1, 3, or 7 bytes remain). Note that receiving equipment would generally not distinguish between sovereign state identifiers and standard identification numbers; they would be treated together only as a single merged number that was known or unknown.

5.2.4 Header Tail

5.2.4.1 Header Tail Organization

The header tail contains one field for indicating the block length in integer format, one field for the standard identification number, and optional fields for error protection, all having been discussed above.

5.2.4.2 Header Tail: Standard ID Organization

The Standard ID comprises two parts: the sovereign identification number, and the standard identification number, described in Sections 5.2.4.3 and 5.2.4.4, respectively.

5.2.4.3 Header Tail: Sovereign State Identification

Eight bits is sufficient to designate the authorizing sovereign state, even if the number of sovereign states exceeds 256; the trick is to subdivide certain undesignated sovereign state identifiers by borrowing bits from the remaining standard identifier field. For example, by deferring to the United Nations the task of defining sovereignty, it should be an easy matter to assign all present U.N. members unique identification numbers in alphabetical order, and to assign the next numbers in order of membership admission to states not replacing member states who are already members. Once 224 member states exist, new U.N. members would share the last 32 numbers. Within these 32 sovereign state "condominium" identifiers, an additional 3 bits would be borrowed from the Standard Identifier field, permitting 8 new member states per condominium, or $256 - 32 + (32 \times 8) = 480$ possible states. Of these state designators, 32 would be assigned to existing standards bodies. In the unlikely event more states are created, still more bits can be borrowed, permitting unlimited growth.

5.2.4.4 Header Tail: Standard Identifier

The Standard Identifier would be selected by the indicated Sovereign State or International Standards Body using existing procedures. To the extent standards already have unique identification numbers, those same numbers could be used here. To the extent they do not, each standards body should map their standards into numbers, preferably a compact or consecutive set. Most new standards might be introduced under long numbers associated with minor standards bodies, while standards achieving wide acceptance could be renamed with short ones. Since bits can be borrowed from the Standard Identifier field almost indefinitely, the system would permit in certain cases (the longer header options) for every individual ever born to become a sovereign definer of standards under some state's authority (the 8-byte option provides each sovereign state with 70 million billion numbers). In the same way, a standards body could allocate numbers well spaced numerically so that their least significant bits could become "user-allocated bits," functioning in lieu of, or in addition to, a descriptor.

Note that the cost of this kind of flexibility, as well as the other sorts of flexibility described earlier, is essentially negligible given the extensible nature of this header architecture. That is, the price of using long headers is paid principally by those who need them, while most users will prefer and use the shorter options.

5.2.5 Header Organization Illustrations

These options can be represented pictorially. The 16-bit composition of the header key consists of three parts: the 4 bits in the length-type LT field (l), the 4 bits in the ID field (i), and the 8 core error-protection bits (p); these protect only the key. The header key always consists of: *lllliiiippppppppp*.

If we designate this 2-byte key by the symbol *K*, and the bytes representing the block length field by *L*, the bytes representing the standard identification number by *S*, and the header parity bytes by *P*, then for LT options 1 and 2 we have only *K* (2 bytes). For LT options 3, 4, and 5, we have only *KSS* (4 bytes), *KSSSP* (6 bytes), and *KSSSS* (6 bytes), respectively. In this case of extremely short blocks, the standard identification numbers *SSS* and *SSSS*, combined with the 4 bits in the ID field, would generally convey messages in their own right, and could also be used for a variety of transactional and transport functions. LT option 7 would permit the following possibilities: *KLS*, *KLSS*, *KLSSS*, *KLSSSS*, and *KLSSSSSSSS*, where the number of bytes allocated to the Standard Identification number by *S...S* is specified by the 3 bits in the ID field (see 5.2.3). One likely option for HDTV might be LT option 12, combined with ID option 1, represented as *KLLLLSPP* and comprising 9 bytes. The standard number contained in field *S* would be an international standard. If such an international standard were devised to have block lengths no larger than 64 Kbytes, and if 1-bit error correction were adequate, then a 6-byte HDTV option is available: *KLLSP*. Such short 5 or 6-byte header options could often be employed for nonvideo information. For example, *KLSS* (5 bytes) would accommodate 64,000 possible international standards employing data up to 256 bytes per block.

5.2.6 Header Architecture: Equipment Implications

Equipment interpreting such headers can be particularly simple. Since it normal-

ly would be reading a stream of blocks, and should know when a block begins, it could simply jump to one of the 64K words specified by the 16-bit header key, these words indicating the location of the bytes specifying the block length. An example of an algorithm to perform these header decoding functions appears in Section 5.2.8, and an illustrative program coded in C appears in Appendix B. Simpler equipment might simply look at the first 4 bits of the LT field, from which the same information can be deduced in an error-free environment. Equipment of intermediate complexity can use the key error-protection bits to an intermediate degree. The ID bits immediately indicate the field where the Standard Identifier is located, and equipment should compare this to a list of standards it is prepared to process. After any indicated processing, the equipment moves directly to the next header at the specified number of bytes along the data stream.

Should synchronization be lost, it could readily be recovered in most cases of interest. For example, in the 9-byte HDTV header example above, these 9 bytes would probably never change within a single broadcast program. HDTV broadcast receivers would simply scan the data stream for that particular 9-byte sequence, which could be used as a traditional synchronization block. This would work even if multiple header types were being interleaved. Accidental synchronizations (very rare) in the data block would be recognizable because the indicated false block length would probably not lead to a valid header.

The most direct use of these header/descriptors would be as a series of bytes at the start of each data block, where the data blocks are then concatenated in a comma-less string of bits. Other uses could also be made, however. For example, a particular transport scheme (e.g., over-the-air HDTV) might package data blocks in discontinuous but fixed form within a larger data stream, this stream being characterized by the embedded header/descriptors. A header/descriptor could also characterize (say) each frame of an HDTV sequence, and the same header/descriptor technique could also be used within each frame to communicate details about its internal structure. Such nesting of header/descriptors does not impair synchronization much, although an initial false synchronization could occur within a larger block; this would be quickly detected if the presumed block ended without a valid header following it. The search for a valid header could then resume.

5.2.7 Header Architecture: Transport Functionality

If one wishes to incorporate synchronization augmentation, extra error protection, block prioritization information, or cryptography for the header or descriptor, it must either be defined at the outset in the header/descriptor definitions, or its incorporation becomes standard-specific — but how could we know the standard if we were not synchronized or error-free, etc.?

For example, in high-error-rate environments users may wish to provide their header/descriptors with more synchronization information or error protection than is available in the basic definitions. Fortunately, much additional transport-related information can be conveyed efficiently through repetition of short blocks. One simple approach is to insert brief bursts of short 2-byte headers into the data stream, where the synchronization powers of these 2-byte header blocks are cumulative; the greater the anticipated channel noise, the longer these bursts should be. Although the software of some receivers may not be sophisticated enough to synchronize such bursts well, this approach is standard independent, and so such software could be designed today. Such a burst inserted anywhere in a sequence of blocks permits synchronization of the entire stream. Since 2-byte headers can correct 2 bits in 16, synchronization bursts fail only when the bit-error rate continually exceeds ~0.2. In still higher noise environments where it is known such 2-byte bursts might be employed, the receiver can either autocorrelate the signal or correlate it with potential 16-bit synchronization words; this could provide synchronization for nearly any bit-error rate, provided the burst length was sufficiently long.

Similar issues arise when extra error protection for the header/descriptor is desired. If the nature of such protection is defined only in a data or standard ID area that cannot be read unless error protection is employed, the data is generally inaccessible. One option is for the user to generate, as above, bursts of short blocks that convey primarily the identity of the desired error-protection scheme employed in the longer data blocks. Such standards could be defined so that they are presumed operative for all following data until "turned off" by another command. Alternatively one of the short blocks (say 2 bytes) could be employed to indicate that error protection, cryptography, or other such schemes were to be conveyed in a "turn-on, turn-off" fashion; a separate short block could be used to convey the opposite message. Such a repetitious series of short blocks can also be well protected and synchronized in essentially any reasonable error environment (BER <0.2) using the existing defined header/descriptor options. In this case the user would need to know only the definition of the chosen error-protection ID number. Such error-protection schemes and protocols can be defined in simple ways over the years.

Yet another similar problem involves the possible incorporation of block priority information. For example, if some data processing, transmission, storage, or display step cannot handle all the data, the originator of that data might wish to tell the user which data is more expendable. Yet the user might wish to do only minimal decoding to determine priority. Although such information would then be descriptor-standard-specific, one or more descriptor standard IDs could be defined in such a way as to convey relative block priority, say on a scale from 1 to 10. Such information could also be embedded in the data itself, but this would generally require the user to do more decoding before discarding any block. Although confusion could arise because multiple priority-labeling descriptor schemes might arise, they would all have standard numbers which, in principle, could be accessed. Section 5.4 describes another approach, which can be used in parallel.

Although this discussion has not been exhaustive, it does suggest that the proposed header/descriptor definition has great flexibility for handling a variety of problems faced when it must supplement or provide transport-layer or other OSI layer functions.

5.2.8 Compact-Header Decoding Algorithm

For simplicity here, we assume the block has been synchronized and that the header key has been error-corrected, perhaps by using a 16-bit dispatch table. We also assume the equipment is provided with a list of standard IDs that it knows how to interpret, as well as a much shorter list of ID length fields corresponding to these standards (possible values of K, defined below).

1. Dispatch on the first byte (256 options); return with 4 integers:

   I = length of length field (0+, 0−, 1, 2, 4, or 6 bytes)

   J = block length B bytes if I = 0 (2, 4, or 6 bytes)

   K = length of ID field in tail (1, 2, 4, 8, or 32 bytes)

   L = presence of descriptor (0 or 1)

2. If I = 0, set block length $B = J$

3. If I = 0−, dispatch (Table 1; not presented here) on M, bits 5 to 8 of the header

   Interpret resulting message and go to 12, or skip directly to 12 if message unknown.

4. If I = 0+, dispatch (Table 2; not presented here) on bits 5 to 8 of header.

   Interpret resulting message and go to 12, or skip directly to 12 if message unknown.

5. If $I \neq 0$, read I bytes (yielding block length B), starting at bit 17 of header.

6. If ID field length K not on list of known ID field lengths, go to 12.

7. Read ID field of length K bytes, starting at bit $17 + 8I$.

8. If ID is not on list of known IDs, then go to 12.

9. If $L = 1$, read descriptor length D (part of descriptor decoding algorithm, not described here).

10. If $L = 0$, then $D = 0$.

11. Go to algorithm specified by ID and execute over a block of B bytes, starting at the end of the descriptor at $I + K + D$ bytes.

12. Jump to end of block (B bytes long) and read next header.

5.3 Descriptor Specification

5.3.1 Introduction

Publicly readable descriptors may or may not be incorporated in any data block, as indicated by one of the bits in the header key. They would convey auxiliary information concerning the nature of the associated data, such as authorship, distributorship, ownership, intellectual property restrictions, sampling patterns, filtering employed, color, nonlinearities, etc. This information would generally be in the form of identification numbers assigned by standards bodies, although options for conveying text, programs, or other data would exist. Like the header, descriptors would indicate their length so that equipment could skip past if desired, and they would have optional provisions for error protection. An efficient ASN.1 equivalent for the descriptor definition proposed below could also be developed.

5.3.2 Descriptor Architecture

The descriptor is divided into three parts: a 2-byte "key," a "core" ranging from 0 to 8 bytes, and the "tail" of length defined by the core. The key unlocks the core, which defines the length of the descriptor, an indication of the nature of the contents of the descriptor, and the nature of any optional error protection for the core. The tail consists of a series of descriptor identification numbers, similar in concept to the standard identification numbers provided in the header. For each camera type, nonlinear luminance mapping, movie producer, filtering algorithm, royalty payment procedure, etc., there could be a separate descriptor number assigned or registered by appropriate standards bodies, indicated in a manner also similar to that of the header. To properly convey this list of descriptor identification numbers, the tail also contains fields giving the number of such descriptor standards, the length and type of each such standard number, and the nature of any optional error protection employed. The tail also supports delivery of fields of text in any of a large number of languages, such as English or Portuguese, as well as computer languages such as C, Postscript, etc. Such software elements would permit the decoding procedures to be defined explicitly, if desired.

A potential area for future improvement is development of a more universal subset of descriptor elements for widespread usage. It would include parameters such as resolution, raster definition, bit packing, etc.

5.3.3 Descriptor Key Definition

5.3.3.1 Descriptor Key Architecture

The descriptor key consists of three parts: (1) a 4-bit "type" field that characterizes the contents of the descriptor, (2) a 4-bit "length type" field that indicates the format of the descriptor length field, and (3) an 8-bit "protection" field for the key.

5.3.3.2 Descriptor Key: 4-Bit "Type" T Field

The purpose of the T field is to indicate: (1) whether or not this descriptor contains a public index (1 bit); (2) whether the descriptor length field in the core (if any) is 2 or 4 bytes long in integer format (1 bit); and (3) which of four error-protection options are being employed (2 bits). The four descriptor error-protection options are: (1) no protection, (2) protected core plus unprotected tail, (3) both core and tail protected, and (4) both core and tail doubly protected.

5.3.3.3 Descriptor Key: 4-Bit "Length-Type" DLT Field

This field contains the length of the descriptor after the core, in bytes, unless its contents are "zero, zero, zero, zero" (for descriptor tails longer than 16 bytes), in which case the length is given by the core in a field which is either 2 or 4 bytes long, as specified in the T field (see 5.3.3.2).

5.2.3.4 Descriptor Key: 8-Bit "Protection" P Field

The function of the P field is identical to that of the 8-bit error-protection field of the header key; it protects the 2-byte descriptor key only.

5.3.4 Descriptor Core Definition

5.3.4.1 Descriptor Core Architecture

The descriptor core consists of three parts: (1) a field defining the descriptor length (0 to 4 bytes); (2) a field indicating the nature of the contents of the descriptor (0 or 2 bytes); and (3) an optional protection field for the core only (0, 1, or 2 bytes). The total length of the descriptor core thus ranges between 0 and 8 bytes.

5.3.4.2 Descriptor Length Field

This field is of length zero if the descriptor length specification has been preempted by the length field in the descriptor key; otherwise it is either 2 or 4 bytes long in integer format, as determined by one of the bits in the T field of the descriptor key (see 5.3.3.2). The length of the descriptor field as presented in the core is defined as including all the bytes in the descriptor, including those in the key, core, and tail.

5.3.4.3 Descriptor Core Contents Index

This field contains 0 or 2 bytes, as indicated by one of the bits in the descriptor key T field (see 5.3.3.2). In its 2-byte form it indicates whether or not the following descriptor contains information concerning any of 16 categories of information about the data stream. Among others, these categories of information include synchronization reinforcement, error-protection data, encryption keys, packet priorities, authorship, distributorship, time or date of any event, ownership, intellectual property restrictions, sampling patterns, filtering history, color, nonlinear mappings employed, etc. The last bit of the index is zero if this descriptor is the same as the previous one associated with the same header ID. The purpose of this contents index is to spare equipment the burden of decoding descriptors when their contents may be of no interest. This is particularly so when a long sequence of descriptor elements is repeated periodically to aid certain users having only segments of the data stream available to them. Users of longer segments could therefore ignore such data more readily.

5.3.4.4 Descriptor Core Parity Protection

This field would contain 0, 1, or 2 bytes, as indicated by two of the bits in the descriptor key T field (see 5.3.3.2). These bytes would protect the core only, using codes similar to those employed in the header.

5.3.5 Descriptor Tail Definition

5.3.5.1 Descriptor Tail Architecture

The descriptor tail consists of four fields:

1. The element-number field, which indicates the number of independent descriptor identification numbers contained in this descriptor; its length ranges from 4 bits to a maximum of 2 bytes.

2. The descriptor element length-type field, which specifies the lengths of each of the descriptor identification numbers contained in the following field, together with their respective types; these types include identification number types similar to those employed for indicating standard numbers in the header, as well as supporting transmittance of text and computer programs; its length is typically 2 to 4 bits per descriptor identification number.

3. The descriptor identification number field, which consists typically of one or more descriptor standard numbers, each in a 1 to 6-byte format or appearing as a sequence of text or code; the total length of this field approximates several bytes per descriptor element, or substantially more if text or code is incorporated.

4. The protection field, as defined jointly by the protection option indicated in 2 of the *T* bits (see 5.3.3.2) in the descriptor key combined with the indicated descriptor length; longer descriptor lengths would require more bytes of protection for any indicated level of protection.

5.3.5.2 Descriptor Tail Element-Number Field

This fields consists of one, two, three, or four 4-bit words indicating the number of descriptor elements contained within this descriptor. Each 4-bit word contains 3 bits indicating the number of elements, and 1 bit indicating whether or not an additional 4-bit word is appended, up to a maximum of four words. Thus one 4-bit word will suffice for 0 to 7 descriptor types, which normally should be sufficient. Two concatenated 4-bit words offer up to 2 exp 6 = 64 possible elements. Three words can accommodate up to 512 elements, while use of all four words (2 bytes) can accommodate more 8000 elements, which should be sufficient and is the maximum number per header allowed under this protocol.

5.3.5.3 Descriptor Tail Element Length-Type Field

This field consists of a series of extensible 2-bit words, one sequence of such words applying to each descriptor element. In most cases a single 2-bit word would suffice; the options here are that a 1, 2, or 3-byte field is reserved for the associated descriptor identification number; the fourth option is that two 2-bit words are being employed. If the second 2-bit word is employed, the associated options are that 4, 5, or 6 bytes are being employed to indicate the associated descriptor identification number; this accommodates up to 10 exp 12 possible identification numbers, which should be adequate. The fourth option available for the second 2-bit word indicates that an additional 4-bit word is to be interpreted. This 4-bit word offers 16 additional options, the first of which is that following the 4-bit word, a 1-byte word specifies the length of the descriptor identification field.

The remaining 15 options are of similar form, but indicate that types of descriptor data are being employed other than the standard identification number type. For example, type 2 would indicate that ASCII text was being employed in a language indicated by the first character of the text stream; thus 256 possible languages can be used. Types 3 to 16 would indicate which of several possible computer languages or image description formats were being employed, such as C, Postscript, etc. If it is felt that the 15 possible languages available under the 4-bit extension option in the element length-type field is inadequate, then additional 4-bit fields could be appended by using an extension bit, or by using 1 bit in each 4-bit field to indicate additional 4-bit fields are appended and can be interpreted as were the series of descriptor tail element-number 4-bit words. (Alternatively, the 4-bit word could be reduced to 1 or 2, with the understanding that the language is specified by the first following 2 bytes.) Definition of these options is left to the next step in the SMPTE standards definition process.

5.3.5.4 Descriptor Tail Standard Identification Numbers

If one or two 2-bit words have been previously employed to indicate the length of the descriptor identification number, than 1 to 256 bytes may be employed for the ID number itself. The formats for each of these options are indicated below.

1 byte	International standard established by single designated authority
2 bytes	16-bit standard number designated by authorized international standards body (bodies)
3 to 256 bytes	1 byte indicating the sovereign state, and the remainder (2 to 255 bytes) being available for the standards number

The sovereign state and standards numbers would be designated using procedures similar to that specified in the header. Note that the longer standards numbers permit subdesignations under the sovereign state indicator for subsidiary standards bodies, including individual corporations, institutions, and even individuals. The longer fields also permit use of user-defined bits that can be assigned at execution, thus providing a data field. Such data fields could be used for conveying dynamically changing information such as average luminance, audio gain, etc.

5.3.6 Descriptor Tail Error Protection

This field could be concentrated at the end of the descriptor or distributed throughout to simplify processing. The number of bytes and protocol employed for this purpose would be determined uniquely by the 2 bits in the descriptor key *T* field and the descriptor length, as specified in the descriptor core descriptor length field or in the descriptor key length field. Definition of these protection strategies might parallel those employed in the header and remain to be defined more fully.

5.4 Transport Header/Descriptors

5.4.1 Motivation and Objectives

Section 5.2.7 discusses several reasons why providing error protection, synchronization reinforcement, packet priority, and higher level encryption to header/descriptors could pose problems for interpretive hardware if the data is excessively noisy. Although the solutions suggested there will accommodate most error environments, still more serious situations can be handled using a transport header/descriptor block such as described here. Such a transport block has the additional advantage that if insufficiently protected data is moving into a more hostile transport environment, additional protection can be incorporated in the transport block without having to redefine the input blocks. Similarly, such transport blocks can be removed without penalty when moving into more error-free or otherwise benign environments.

The objective here is to suggest how the architecture of such transport blocks are consistent with the header/descriptor definitions presented above, but not to define all the details. Thus establishment of the header/descriptor standard can proceed without waiting for all details of the transport block to be resolved. It would be useful to resolve such details, however, before users of the standard adopt inferior methods of addressing the same transport problems.

The principal motivation for defining transport blocks is to avoid proliferation of standard-specific alternatives for addressing such transport problems. Such proliferation could increase the cost of decoding equipment that would have to accommodate all these possibilities. In a high-error environment, executing the multiple search strategies necessary when synchronization is lost or heavy errors exist, could become prohibitive. For this reason it is important to have only a few standard options for certain aspects of the transport block. Defining an efficient small set is beyond the scope of the present effort and would take considerable study. Therefore it is reasonable to assume that this study would be completed after any initial header/descriptor standard is specified. One illustrative candidate for such a set appears here in Appendix A; it is intended only to initiate discussion of these issues.

5.4.2 Architecture of Transport Blocks

Transport header/descriptor blocks would consist of a single header/descriptor, where the header would specify an international standard ID indicating which type of transport block was involved; only a few such types would ever be defined. The descriptor of the transport block would convey up to eight different elements:

1. Descriptor table of contents (standard format defined earlier).
2. Synchronization reinforcement bits, not error protected.
3. Error-protection bits for the transport header and its attached following

header/descriptor.

4. Encryption key for deciphering the descriptor, if any, in the following block.

5. Block priority; determined by data originator, indicating relative priority of the following block concerning interpretation or transmission in cases where inadequate capacity is available. Authorization keys may also be needed to verify priority in certain cases. Price bidding could be supported here too.

6. Authorizations and fee mechanisms for alteration or use of data.

7. Block sequence numbering and timing-reconstruction information.

8. Padding to yield one of the very few allowed lengths for the transport block corresponding to the international header standard ID.

In addition to these elements there would also be the traditional field in the descriptor defining its length, although interpretation of this length would be unnecessary because the transport lock length is specified by the header, and there is no data payload following the descriptor.

The six main descriptor elements would convey information using traditional descriptor standard numbers, where long numbers accommodating an adequate number of user bits could be employed. Defining these descriptor standard numbers is the task that can and probably must be postponed until the technical tradeoffs associated with different choices are better understood. Thus standards for headers and descriptors can and should be adopted in advance of these descriptor definitions for transport blocks.

5.4.3 Decoding Issues for Transport Headers

Decoding such a transport header in a high-error environment would be relatively straightforward. First, if synchronization had been lost, synchronization would be established. Initially this might be done assuming a low-error environment. If the environment is noisy, then each of a few possible synchronization blocks would be sought, including periodic repetition of legal header keys, assuming that key bursts might have been employed for this purpose. Because all possible synchronization words might be sought, it is important to have only a few legal ones if they are many bytes long. Because the sync reinforcement bits could be only in a small number of positions relative to the beginning of the transport block, each such position could be tested for consistency with the associated error-protection bits, which are also located in a small number of possible locations. Confirmation of synchronization follows if legal headers come immediately after the indicated end of any block. Once the transport block is synchronized and error corrected, the remaining descriptor fields containing any encryption key for the descriptor in the following block, or any packet priority information can be deciphered.

One principal new constraint should be imposed by the standard on manufacturers of equipment handling this header/descriptor standard. If transport blocks are to be useful, such equipment must never insert data between a transport block and the following block to which it applies. Transport systems should try not to scramble the sequence of data blocks in any event, but if a transport block should accidentally be prepended to the wrong data block, the packet priority, encryption key for the data block descriptor, and the error protection for the header/descriptor could be inappropriate, resulting in a scrambling of the interpreted header/descriptor. Such scrambling would also typically cause local loss of synchronization, particularly in high-error environments. Since transport blocks would normally be quite brief compared to typical data blocks, such a constraint should not be difficult to satisfy. Such transport blocks could also be added or subtracted at will by a given transport layer, however, provided they are appropriate to the blocks which they precede.

If a data stream is entered in the middle of a transport block, then confusion might result. To protect against this unlikely possibility, equipment might choose to wait until the second valid header is intercepted before commencing decoding.

6. Abstract Syntax Notation 1 (ASN.1) Header/Descriptor Architecture

6.1 Background

Abstract Syntax Notation 1 (ASN.1) is an existing ISO/CCITT Standard in common use within the computer and telecommunications industries. Within the ASN.1 framework, it is straightforward to define an SMPTE header/descriptor that meets the objectives described in Sections 1 to 5 above, and it would leverage existing tools, expertise, and administrative structures.

ASN.1 is derived from earlier work at Xerox PARC on Courier (late 1970s). An early version of the notation (c. 1984) was used in the first draft of the CCITT X.400 series of recommendations on message handling systems (i.e., electronic mail). ISO and CCITT then jointly developed ASN.1 for use within the OSI presentation layer (c. 1988).

ASN.1 is now widely used in a range of standards activities, including the CCITT X.500 directory service and both the OSI and Internet network management systems. Over the years, a collection of software tools and utilities to support ASN.1 has been (and is being) developed.

6.2 Concepts

ASN.1 is an extensible notation for describing data that is to be exchanged by transmission or storage. It is much like a programming language, such as C and Pascal. There are several simple types, such as integer, real, and octet string (i.e., byte string), and constructor types that can be used to build arbitrarily complex data structures, including hierarchical representations (e.g., packet within packet).

An ASN.1 header can be thought of as an envelope that contains, for example, a single video frame. ASN.1 supports the notion of embedding, which allows one or more data structures to be contained within another. Thus, a sequence of frames can be embedded within an outer header (or envelope) that labels a program segment. This can be taken to coarser granularity, e.g., shots, scenes, programs, etc. Similarly, it can be taken to finer granularity to embed audio tracks, closed captioning, descriptors, etc., within individual frames.

A key feature in ASN.1 is the separation of how the data is described (Abstract Syntax) and how data is encoded (Basic Encoding Rules, or BER). Data structures are described in a human-readable syntax and automatically translated into the bits and bytes for transfer. When a new data structure (or type) is defined, its representation is automatically generated. Furthermore, deployed ASN.1 compliant systems will be able to interpret new structures without hardware modification.

The following summary description of ASN.1 presents only enough detail to motivate its use for the specific needs of a header/descriptor. For formal definition of ASN.1, refer to ISO 8824/8825 and/or CCITT X.208/209. A more accessible description can be found in: Marshall T. Rose, *The Open Book: A Practical Perspective on OSI,* Prentice-Hall, 1990.

6.2 Basic Encoding Rules (BER)

All ASN.1 types, whether a simple type or a structured type, can be encoded using the same basic format of three fields:

[tag] [length] [value]

The three fields together make up a data item. Each field is variable in size to accommodate arbitrarily complex substructures and encodings. A simple type, such as an integer, requires only a few bytes. A structured type, such as a long byte string, can be Mbytes, Gbytes, or larger as necessary to contain the payload data value. The basic format is inherently self-identifying and extensible.

A data stream is a sequence of items, each of which can be structured or nested. Thereby, one can define arbitrarily structured data for both header and descriptor, including nested packet-within-packet structures.

Tag Field. The tag field specifies the type of the item value. Several simple and structured types (integers, character strings, etc.) are universally defined in the ASN.1 standard, and are recognized in all

compliant environments. Also, one can define types that exist within the specific environment of an application or communications context. A tag is principally encoded as a single byte, but can be extended. In the BER, the tag is encoded as:

<2> <1> <5> --bits/field
[class | p | tag number]

The tag field allows a receiver to "parse" the incoming data stream, selecting those components/types in which it is interested and bypassing others.

The EXTERNAL type is of particular significance. In essence, it is a universal header for the data that it encapsulates. The EXTERNAL type is described further below. Other types are relevant to use in a descriptor.

Length Field. The length field indicates the size of the value. It is an integer of one or more bytes that specifies the number of bytes in the value field. In the BER, the short form of length is encoded as a single byte, and can indicate lengths of 0..126 bytes of value. In extended form, the first byte specifies the number of bytes of length. Length is encoded as:

short form : 1 byte : [0bbbbbbb] : lengths of 0..1126 bytes

extended form : n bytes : [1nnnnnnn] [bbbbbbbb] [bbbbbbbb] ...n

Only the number of bytes needed to specify length are used. Thus, the length field is compact. The extensibility of the length field permits a maximum length field that is 126 bytes to specify a length of ~2^1008 or ~10^303.

Note that a common length specification is used regardless of the associated data item. Accordingly, there is no need to invent custom length encoding schemes for each new data item.

Value Field. The value field is the value in the type specified by the tag. It is a string of bytes the number of which are specified by the length field. It is encoded as defined in the BER for that type.

6.4 ASN.1 EXTERNAL Type (Universal Header)

An ASN.1 EXTERNAL type is a universal header. All ASN.1 compliant protocol interpreters can extract and interpret an EXTERNAL without ambiguity. The definition of EXTERNAL is quite flexible, but that flexibility is not needed here to meet the basic objectives of a universal header.

A simplified EXTERNAL type is encoded as:

[tag = class = 0,p = 1,number = 8] [length] [object id] [payload]

Tag and **length** fields are encoded as described above. **Object ID** provides unambiguous self-identification for the header. **Payload** is a sequence of bytes that are interpreted according to the standard indicated by the object ID.

Object ID is itself an ASN.1 type with tag, length, and value fields, and is encoded as:

[tag = 0,0,6] [length] [id value]

Object ID value is a sequence of bytes that represent the hierarchical identifier for the referenced standard. ID values are assigned, registered, and administered by CCITT and ISO in the course of standards development. Or, ID assignment can be delegated to member bodies or companies or organizations (thereby, SMPTE could assume responsibility for administering a portion of the ID space.) The root prefix values are:

CCITT [0]		
recommendation	:	CCITT committees
question[1]	:	CCITT Study Groups
administration[2]	:	country Ptts (country code)
network operator	:	X121 organizations
ISO[1]		
standard[0]	:	ISO standards
registration authority[1]	:	ISO authorities
member body[2]	:	member bodies (country code)
identified organization[3]	:	organizations
joint ISO CCITT[2]	:	assignment delegated to ANSI

A few prefixes are of particular note. iso.standard registers all ISO standards. ccitt.administration and iso.memberbody are assigned to sovereign bodies (identified by their international telephone country code). iso.organization is assigned to international organizations. These cover virtually all the situations under which a header identifier will need to be assigned.

An ID value is encoded as a sequence of bytes. The first two levels are encoded in the first byte — $a.b. = 40*a + b$, $a <= 3$, $b <= 39$. Remaining levels are encoded as one or more bytes as needed to represent the numerical value for that level. If a value is greater than 127, it requires more than one byte; the MSB of the byte is set to indicate that the value is continued in the next byte, and so on. For example, iso.standard.jpeg is 3 bytes:

iso.standard.jpeg :: 1.0.10918*
= [40] [128+85] [38]

Similarly, Group 3 Fax is identified as:

ccitt.recommendation.t.4 :: 0.0.20.4
= [0] [20] [4]

The extensibility of the ID value field permits up to 126 byte long IDs to specify distinct IDs numbering to ~2^882 or ~10^265.

Payload is an ASN.1 type with tag, length, and value fields, encoded as a sequence of bytes that are interpreted according to the standard indicated by the **Object ID**.

[tag = 2,0,1] [length] [payload value]

6.5 ASN.1 Descriptor

In addition to the EXTERNAL type (for use as header), ASN.1 defines other basic types to represent a variety of values, including: booleans, integers, reals, byte strings, character strings, universal time code, etc., and constructions of values into arbitrary data structures.

Although a distinction is drawn between header and descriptor in this report, the ASN.1 approach permits a single mechanism to serve both functions. The full benefits of ASN.1 become apparent when it is applied to the descriptor. Especially the ability to construct new types and to incorporate references to other standards. (See current ISO work on Image Interchange Format [IIF] for an example of ASN.1 use in defining descriptors.)

6.5 How ASN.1 Addresses Objectives

The following describes how an ASN.1 header/descriptor addresses the objectives stated at the beginning of this report.

Universality — ASN.1 header/descriptor promotes and enhances universality:

- It complies with and recognizes existing standards and practices. All existing and future ISO/CCITT standards are uniquely identified by an ASN.1 Object ID.
- It addresses the issues of sovereignty. ISO/CCITT administers and delegates assignment of Object IDs to subcommittees, member bodies (by country code), and organizations. The complexity of this task and the benefits of leveraging existing administrative structures should not be underestimated.
- It facilitates coordination among television, telecommunications, and computer industries.
- It sets a minimal level of compliance for low-cost receivers. Furthermore, the advanced stage of definition, tools, and expertise will facilitate the rapid deployment of header-compliant devices.

Longevity — ASN.1 has inherent longevity:

- All fields of ASN.1 types (tag, length, value) can be extended. Payload lengths from a few bytes to ~10^303 can be represented. Similarly, Object IDs can range from a few bytes to ~10^265 bytes.
- It has a preexisting registry and is self-maintaining. ISO/CCITT already registers, administers, and delegates assignment of Object IDs.
- It defines immutable identifiers. Once an Object ID is assigned, it exists "for all time." In the future, when an old

*At this writing, JPEG is nearing but not yet an ISO standard. Thus, though thought to be correct, the number here (1.0.10918) is not yet official.

ASN.1 header is recognized, there is no ambiguity to the referenced standard.

Extensibility – ASN.1 is inherently extensible:

- All fields of ASN.1 types can be extended without redefinition.
- New types can be defined and their encoding automatically generated without the need to introduce new rules.
- Since ASN.1 is fully defined, any compliant receiver and equipment are guaranteed to be able to recognize future extended ASN.1 headers.

Interoperability – ASN.1 is inherently interoperable:

- It has a well-formed public definition.
- It is already in use in several important applications and industries.
- It complies with existing and developing standards (including image standards).
- It allows standards and structures to cross-reference each other. The video data stream can contain structures defined by other standards, and vice versa.
- It permits the same information to be interpreted in different ways within different domains without prejudice. To the video industry, the information is a continuous video stream. To the telecommunications industry, the same information is a sequence of bits to be transmitted. To the computer industry, that sequence of bits is interpreted as data structures.
- It permits independent formal definition of intellectual property protection, encryption, and other source coding descriptors by appropriately sanctioned expert groups (either ad hoc or extant).
- Experts groups and standards bodies can autonomously develop and evolve specifications within the domains of their expertise. The ASN.1 cross-referencing capability achieves a degree of interoperation and coordination among parallel activities – coding details are automatically resolved, avoiding redundant and/or conflicting efforts.

Cost/performance effectiveness:

- It is straightforward to recognize and decode with a single uniform procedure.
- Uniform decode hardware and software can be shared among industries and applications yielding economies of scale.
- Existing tools and expertise can reduce the time and cost of deployment.

Compactness:

- ASN.1 is compact, but not so much as to complicate decoding or to compromise other objectives. A typical ASN.1 header would be ~15 bytes for a 1-Mbyte payload (i.e., ~.0014% overhead).
- A 7-byte ASN.1 header can be realized using the standard's indirect reference option. (In fact, a 2-byte context specific header may be realizable.) Given typical payload sizes, however, it is unlikely that this level of compactness will be required – small payloads are either aggregated or infrequently used, especially in the video domain.

Rapid Capture:

- An ASN.1 header is signified by its first byte tag field (equal to 8) followed by length, object ID, and payload. Thus, an ASN.1 header is straightforward to recognize in the data stream.
- ASN.1, like any length/identifier header, provides early identification of the payload.

Editability:

- ASN.1's structuring capabilities permit arbitrary editing, sequencing, structuring, and embedding of payload streams. All of this is accomplished within a single uniform mechanism, and without requiring unnecessary decoding of the payload itself.

Appendix A – Transport Header

A.0 Introduction

The header-descriptor design provides for binding of a transport header to the main header. This is accomplished only by mandating that the transport header not be separated from the main header and its associated data, which it is transporting. The function of the main header is to identify the data that follows. The function of the transport header is to help the main header and possibly its payload in its journey from origination to destination.

The following transport header design is "work in progress," and therefore is meant to be an example rather than the final structure. It supplements the discussion in Section 5.4. The design issues and interactions are a bit complex, so the principles of the transport header design are best illustrated by way of a correctly designed example. If adjustments are made to this design, then care must be taken concerning effects of such adjustments to the rest of the design and the functioning of the transport header. In particular, there is a rigid requirement that certain fields be prespecified as to length, or as to length-specification (type fields) in their meaning.

It is necessary that the transport header be totally independent of any standards described by the main header. This is required because it may be necessary to change transport characteristics for all main headers on a given data stream, irrespective of the standards or formats of those main headers.

An example might be the need to provide improved error protection when moving a data stream from a high-reliability fiber to a high-error-rate radio frequency transmission.

A.0.1 ASN.1 Transport Header Yet to be Designed

The transport header design example illustrated here works together with the header/descriptor design, as described in Sections 1 through 5 of this SMPTE report.

No transport header has yet been designed for the ASN.1 syntax, illustrated in Section 6. Thus, the ASN.1 syntax is, at present, only suitable for error-free channels that preserve the data and its order without contention from source to destination. In order for the ASN.1 syntax to meet its ojectives of interoperability with imperfect or congested channels or media, a mechanism similar to the transport header example shown here will be required. It is hoped that a transport header design might be developed for the ASN.1 syntax system, possibly modeled on the transport header illustrated here, together with the main header/descriptor design. The enormous flexibility of ASN.1 syntax must be tempered to provide a limited number of options for transport headers, each with appropriate protection/correction mechanisms. It is hoped that registration rules and flexibility in ASN.1 can also be used to provide a suitable format transport header design that is properly restricted. Byte alignment is also part of the structure of length fields. If bit alignment is needed for ASN.1, then further suitable adjustments will be required.

Another method to provide transport assistance is to convert from the main header/descriptor design of Sections 1 through 5, to and from the ASN.1 notation. Since a transport header design is available for the main header/descriptor, conversion to this header would provide access to a transport header. When reentering error and contention-free environments, the header/descriptor could be reconverted to ASN.1 syntax.

If the ASN.1 method becomes popular, then it is hoped that a suitable ASN.1 transport mechanism might be developed.

A.1 Design Objectives for the Transport Header

- The transport header must be standard-ID independent, so that it can apply to all header standards equally and uniformly across the entire data stream.
- The transport header should be removable without damage to the function of the main header that it is helping to transport.
- A transport header must be able to be added to any header format or descriptor format without changing any of the meaning.
- The transport header formats should support "in the clear" protection of the main header and payload, where the bits are not altered, so that the transport header can be added and removed without any adjustment of bits within the main header, its descriptor, and its payload.
- In addition to support for "in the clear" protection, more

efficient protection should be supported (such as Reed-Solomon), where the header, and possibly its payload, are encoded.

- The transport header architecture should support one or more mechanisms for correcting burst errors.
- Optional support should be provided for encryption of the main header's descriptor, as well as the data payload.
- The transport header should support authorization and use information, in helping the transport system determine which destinations are appropriate for further or final distribution.
- The transport header architecture should support backward play through the data stream.
- The transport header should support optional rapid header synchronization capture.
- The transport header should support communications networks by providing information concerning the data's priority and value.
- The transport header should support timing reconstruction when utilizing networks that distort timing or ordering of data.
- The transport header should support data ordering requirements, when utilizing networks which might reorder the data.
- The transport header architecture should strike a balance between the opposing forces of flexibility and ease of use. Thus, a small number of options should be carefully chosen for maximum flexibility, with the small number of options allowing a simple interpretation. It is the fact of having a small number of options that allows easy interpretation.
- For transport functions that encode the header and/or its data, a simple "in the clear" length field should allow devices that cannot decode such data to skip to the next transport header.
- For devices that cannot process the transport header, a simple "in the clear" length field should allow such devices to skip directly to the header. Further, such devices should be able to easily interpret the transport header format so they can quickly determine whether the main header is "in the clear," and therefore directly readable.

The following example design meets these objectives.

A.2 Three Types of Transport Header Allocated in the Header Key

In order to meet these objectives, there are three types of transport headers. The first is the basic transport header, which provides the majority of transport capabilities. The second is the redundancy transport header, which is used to protect against burst errors. The third is the reverse transport header, which is used for reverse play.

Three of the 2-byte header key's 256 possible codes are required for the transport headers. In the current 2-byte header key design, 1 byte is dedicated to error protection. The other byte is split into two 4-bit fields.

The first 4-bit field is the "length type" field. Codes 15 and 16 (numbers 14 and 15) are unallocated. These two codes enable the second 4-bit ID field to be used for special purposes such as designating the three transport header types. Thirty-two such codes are available, leaving 29 codes unallocated if 3 codes are assigned to the three transport header types. The basic transport header provides most of the capabilities required.

A.3 Functions of the Transport Header

The functions of the transport header are as follows:

1. Sync reinforcement for those data transport media where it is desirable to rapidly or simply sync to the headers or header-data combinations on switching between streams.
2. Improved error protection via extra protection bits for both the transport header and the main header and its descriptor.
3. Conveyance of priority for the main header and its data, for those cases where a channel may be operating at capacity and thus where channel controllers must decide which headers and their payloads it must drop. Authorization keys may also be needed in order to verify priority. Network accessing methods, such as pricing-bidding techniques, would also be supported here.
4. Encryption and security information for the main header's descriptor, possibly combined with the descriptor's own encryption and security information, in order to protect the data stream following the main header. The protection optionally provided by the main header's descriptor may need to be augmented when the data stream is sent through public or vulnerable exposed channels.
5. Authorization information. Such information would indicate who could receive, who could edit and reassemble with other material, etc. Also, copyright and royalty-fee information would be enabled here. Perhaps automated mechanisms of fee for usage would be supported through this field.
6. Sequence numbers may be added where networks are used for transport, which may reorder packets. In addition to sequence numbers, the combination of the transport header may require information from the main header's descriptor in order for the network to be able to guarantee delivery within known latencies and time windows. For out-of-order delivery, some networks can control the amount of time between a given delivery and the delivery of neighboring data. In the case of images and audio, some devices can accept out-of-order information in their buffers or processing units, but the time is constrained to within one or more frames, or fractions thereof.
7. Timing reconstruction. For those applications where exact timing relationships must be reconstructed from a mixed data stream, the transport header and the main header's descriptor would communicate to provide timing reconstruction information.
8. Reserved for future use.
9. Pad. There is a pad field at the end of the transport header in order to make the length of the transport header plus the main header, its descriptor and, optionally, its payload, come out to a length appropriate for the error-correction protection formats supported in item 2.

The construction of the transport header involves a strict ordering of fields as above, so that the sync reinforcement is always first, the error protection is always second, etc. In this way, if each field is present, its location is easily determined. The error-protection field will always be in a known location, and a "scope of protection" within this field defines those fields that are protected in both the transport header as well as in the main header and its descriptor.

A.4 Redundancy Transport Header

The redundancy transport header provides a special function for error protection against burst errors on the data stream. Improved error protection against burst errors is achieved via redundant copies of future and previous headers (these transport headers providing redundancy will not be bound to a main header, and therefore represent an exception to the binding property of transport headers).

Such special redundancy transport headers will "stand alone" and be occasionally interspersed in the data stream. They will contain error-protected copies (via either the efficient or inefficient method) of some future or previous header, together with a pointer to that header, and a number indicating how many headers forward or backward will be traversed before reaching that header. The previous copies are useful for going backward through a data stream, and for reconstructing damaged data streams on physical media, such as disk.

Using a separate header key code, the redundancy transport header has the format shown in Fig. A1.

The transport header key contains a separate special code indicating that this is a redundancy transport header containing a copy of a future or previous header. This field could possibly be followed by a length field, indicating how to skip past this duplicate header.

The pointer to the header being duplicated is analogous to a length field, but it points past several headers to the header being duplicated. It is a signed number so that it can reference previous as well as future headers. Thirty-two bits of protection are pro-

vided. A number of headers forward or backward is provided, indicating the location of the header being duplicated in number of headers rather than via a pointer (length).

The maximum and minimum millisecond tolerance fields indicate the tolerances for separation times between this copy of a previous or future header and the header itself. Information about separation constraints is provided by these fields for channels that reorder, insert, and remove data.

A copy of the transport header is provided, if there is a transport header on the header being copied.

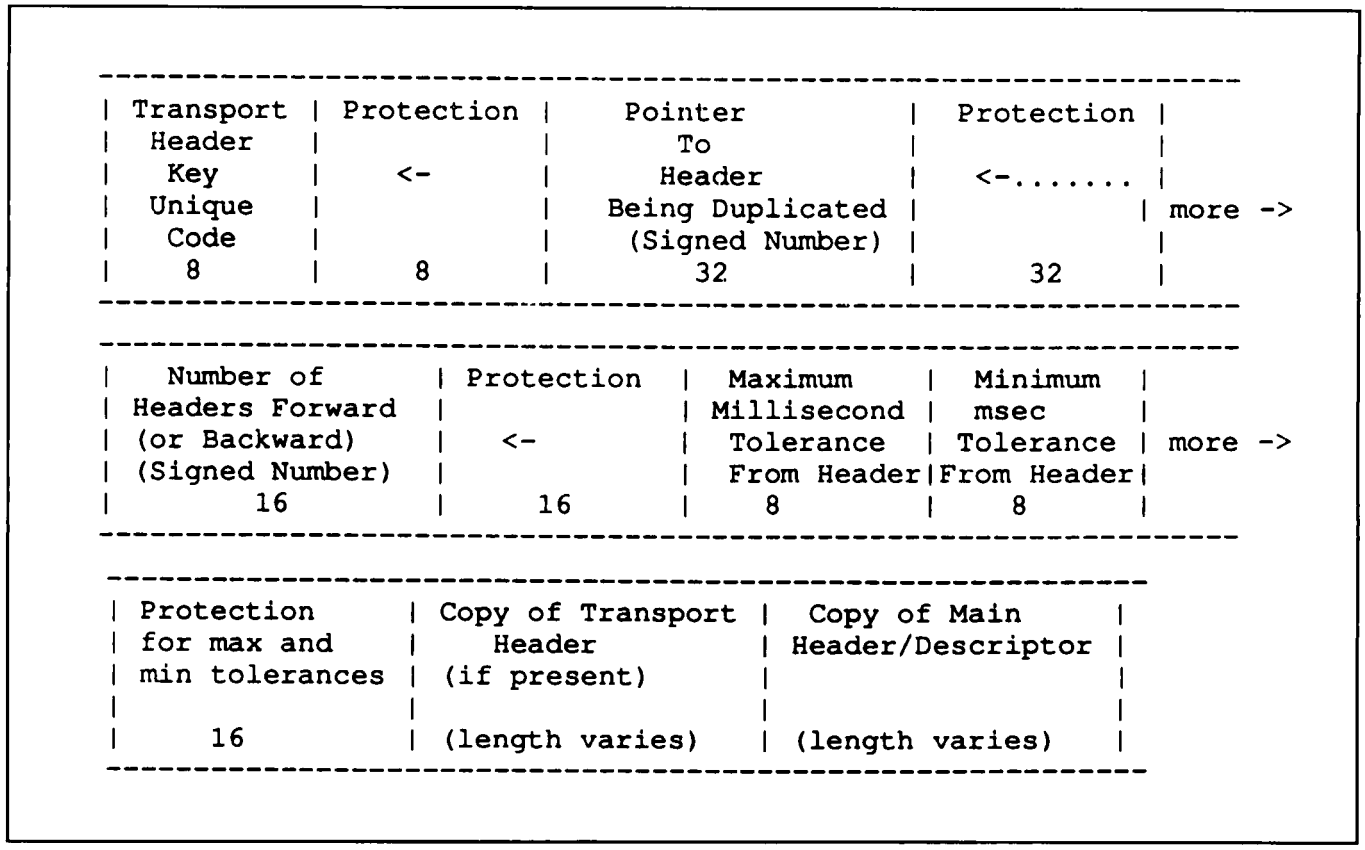

Figure A1. Format of redundacy transport header.

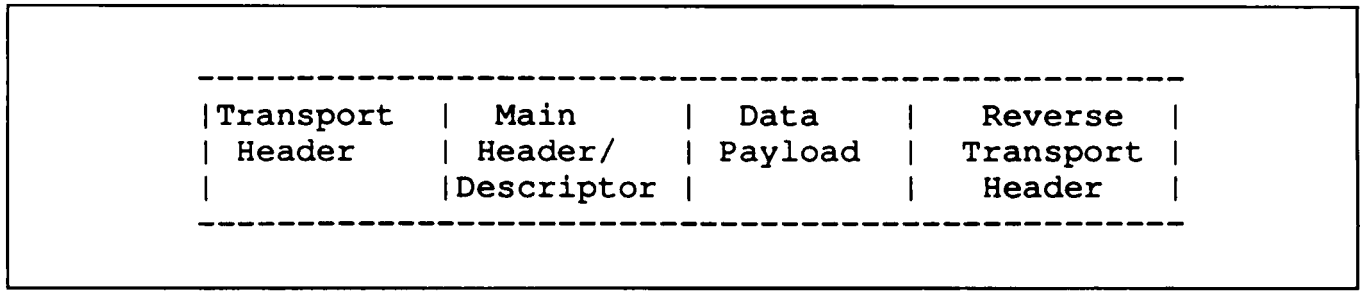

Figure A2. Situation of transport header.

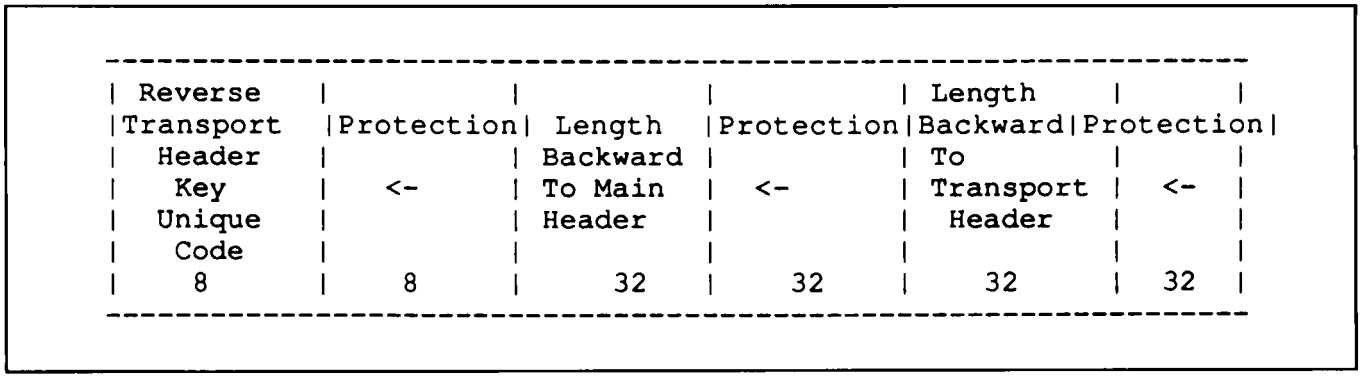

Figure A3. Reverse transport header format.

Finally, there is a copy of the complete main header and its associated optional descriptor. None of the payload is duplicated.

A.5 Reverse Transport Header

In order to support reverse reading of the data stream, a reverse transport header can be utilized. The reverse transport header immediately follows the main header. Thus the transport header is situated as shown in Fig. A2.

If the main header is preceded by a transport header, then the reverse transport header will point back to both the main header and the transport header. If the transport header is absent, then the reverse transport header will point back only to the main header, and the length backward to the transport header will be zero.

The reverse transport header format is shown in Fig. A3. This design allows these reverse transport headers to be appended after the main headers to allow backward traversal through the data stream.

It should be noted that when redundant transport headers are in use to protect against burst errors, reverse transport headers must follow each such redundant header. In that case, the length backward to the main header will be zero, and only the length backward to the transport header will have a nonzero value.

Note that this header has a fixed length. Thus, when encountering this header in the forward direction, no pointer to the next header in the form of a length field is required. The fixed length of 18 bytes is predetermined and can be used to skip to the next header. Also, there will never be a payload of data or any attachment to any headers in the forward direction.

A.6 Transport Headers and Device Capture

It is important to remember that all devices that can edit the data stream must preserve the relationship of the preappended transport header to the main header. Also, the postappended reverse transport header must also remain attached to the main header, if present. Thus, when a transport header is read, the following two headers must also be read before assuming the header and its data and transport have been passed.

When capturing a new data stream, it is necessary to read at least two headers to determine if the first main header is valid. If it had been preceded by a transport header that encoded the main header's descriptor and/or payload, then the data will not be readable without the transport header. Thus, capture is not achieved until the second header has been read, which would accomplish the determination of the first valid header and its transport header, if present.

A.7 Transport Header Format

The transport header format is shown in Fig. A4.

A.7.1 Header Code in the Header Key

The transport header begins with a normal header "key," consisting of 1 byte of key information and 1 byte of protection, as for the main header. However, the transport header uses a header key code reserved specifically for the transport header. The transport header tail differs from the main header tail and has the format outlined in Fig. A4.

A.7.2 Length to Next Header after Main Header

The next field is a 32 bit length field, which points to the next header after the main header attached to this transport header. This length may be required to allow skipping past the main header and its data payload. This will be required by those devices that cannot provide error-protection decoding, when the main header is protected using the types of error codes that scramble the main header. This length is protected by 32 additional protection bits to provide reasonably robust bit-error correction, using the type of correction that does not scramble the 32 bits of length (e.g., Hamming code).

A.7.3 Transport Header Length

This field indicates the length of the transport header. It therefore forms the mechanism to skip to the main header, if there is no desire to read any of the transport fields, and if the main header is not scrambled due to error protection, encryption, or other operations from the transport fields. This field is also protected by a 32-bit field, using a nonscrambling error-protection code.

A.7.4 Type Fields

There are eight 4-bit type fields, to allow 16 types for each of the eight fields in the transport header. These fields are (1) sync, (2) error protection and correction, (3) priority and authorization and bidding for priority, (4) authorization for data use, (5) encryption protection, (6) sequence number and out-of-order timing margins, (7) timing reconstruction information, and (8) a final field reserved for future use. The 32 bits of type fields are protected by 32 bits of protection/correction code.

A.7.5 Sync

The first transport operation field is sync reinforcement. Sixteen types of codes are possible, each with a permanently assigned length. These types and lengths must be permanently assigned from the beginning. A possible set of 16 assignments for lengths might be as follows.

Two types of sync codes could be used, with custom-designed unique spectral signatures. Each of the two types of codes could have one of the following lengths:

2, 4, 8, 16, 32, 64, 128, and 256 bytes

This would result in a total of 14 sync reinforcement patterns. Sync type 0 would indicate an absence of the sync reinforcement field. Sync type 15 could represent an additional special sync field, with a specified length.

It is necessary for all sync type lengths to be specified in advance. Although the codes for sync themselves can be specified later, they can only be specified once for each of the 15 valid sync types.

It should be noted that the transport header always begins exactly 26 bytes prior to the first byte of the sync reinforcement field. Once the sync reinforcement field has been located, the transport header and the main header have been located.

A.7.6 Error Protection

None of the fields previously described, which precede the error-correction/protection field, will be protected by this field. However, all of the fields following and including this error-protection/correction field, starting at the first bit, will be part of the error-protection/correction group. The protection will extend through the rest of the transport header and on into the main header and its descriptor, and for some of the formats, into the payload as well.

Since sync reinforcement is used to capture the data stream initially, it is probably not appropriate to error-protect this sync. The sync codes are designed to be found within a stream. If protection is needed for the sync field, then special sync protection can be provided within the ample bytes available for sync codes.

Two types of error protection are supported. One type allows the protected data fields to be read, as is, leaving them "clear" but augmenting them with protection. Examples of this type of code are the Hamming code and some common forms of the Reed-Solomon code. The second type scrambles all of the bits in the coding process. An example of this type of code is a convolutional one such as the Viterbi code. In the case of scrambled protection, the fields being protected will be completely unreadable without decoding. If the header format, the header length field, the descriptor type, and many other crucial fields are protected in this

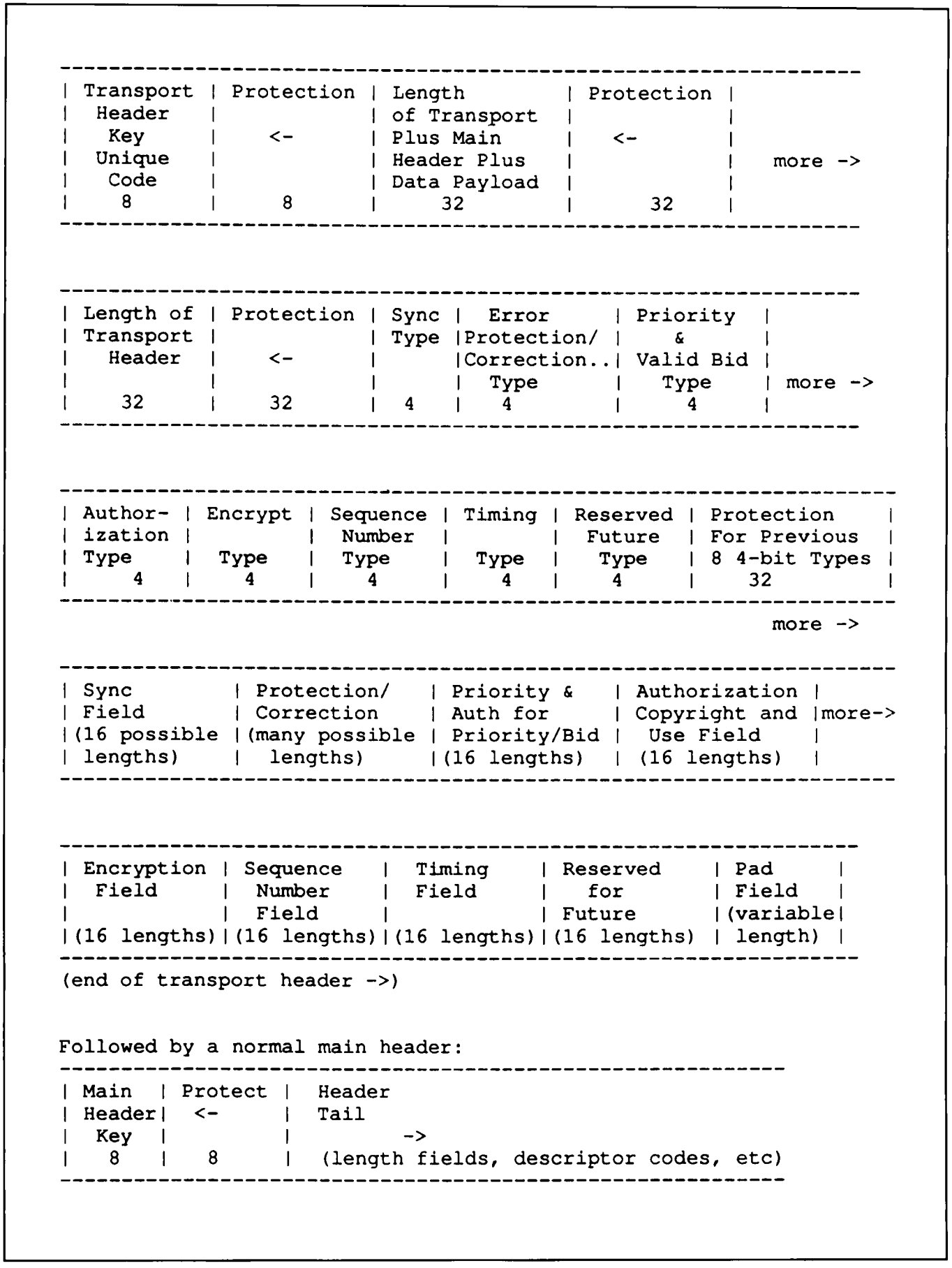

Figure A4. Transport header format.

way, those devices that cannot decode the error correction would be unable to either read or skip the header. Thus, the transport header will contain a "length field," protected in the augmentation method rather than the scrambling method, which will allow devices to find this length field and thereby skip the rest of the transport header and the entire main header, its descriptor, and its payload. A second "length of transport header" field will also be present and protected via the "in the clear" augmentation method.

A.7.6.1 Intentional Inflexibility

In order for all of the above mechanisms to operate properly and efficiently, it is necessary to limit the number of formats available in the transport header. Perhaps 8 or 16 types of each of the fields should be provided for, with all types being specified in advance. We are using 16 as our example. Thus, there would be 16 error-correcting coded formats and {_ code lengths, 16 sync reinforcement field formats and lengths, 16 priority fields and lengths, etc., with each field format having a specified length.

Since the error correction transport function is the most difficult with respect to format, a very limited number of field options will be provided under the scope of protection. Also, padding, which might be quite long, will be required at the end of the transport header before the main header in order to make the total length of the transport header plus the main header and its descriptor (and possibly payload) to be a simple-to-correct convenient known length corresponding to one of the 16 protection types.

A.7.6.2 Possible Prespecified Protection Types

The prespecified 16 possible protection types might be as follows:

- Type 0 indicates that no protection/correction field is present.
- Types 1 through 10 protect "in the clear," by adding protection bits without scrambling, as in Hamming codes and some common types of Reed-Solomon coding.
- Types 1 through 5 protect the transport header, the main header, and its descriptor.

The types are as follows:

1. Protect all remaining transport header bits, beginning at the first bit of the error-protection/correction field, all main header bits, and all descriptor bits. Do not protect any payload bits. The protection/correction bits are applied on every group of 64 bits. The total length of all fields being protected, excluding the error-code bits themselves, must be a multiple of 64 bits. This is accomplished by the use of pad bits at the end of the transport header.

This implementation requires that memory be available to store the correction bits for the entire length of transport header, main header, and its descriptor.

For type 1, the total length of the protection code field is the total length over 16, with 4 bits for every 64 (68 bits total on 64 bits of data).

2. Same as 1, but total length over 8, with 8 bits protecting every 64 (72 bits total for 64 bits of data).
3. Same as 1, but total length over 4, with 16 bits protecting every 64 (80 bits total for 64 bits of data).
4. Same as 1, but total length over 3 (half as long as the fields being protected); 32 bits protecting every 64 (96 bits total for 64 bits of data).
5. Same as 1, but total length over 2 (the same length as the fields being protected); 64 bits protecting every 64 (128 bits total for 64 bits of data).

Types 6 through 10 protect the payload in addition to the transport header, the main header, and its descriptor.

6. Same as type 1, except protect the payload as well. For type 6, the total length of the protection code field is the total length over 16, with 4 bits protecting every 64 (68 bits total 64 bits of data).
7. Same as 6, but total length over 8, with 8 bits protecting every 64 (72 bits total for 64 bits of data).
8. Same as 6, but total length over 4, with 16 bits protecting every 64 (80 bits total for 64 bits of data).
9. Same as 6, but total length over 3 (half as long as the fields being protected); 32 bits protecting every 64 (96 bits total for 64 bits of data).
10. Same as 6, but total length over 2 (the same length as the fields being protected); 64 bits protecting every 64 (128 bits total for 64 bits of data).

Types 11 through 15 protect by scrambling, as in convolution coding techniques such as Viterbi coding. Types 11 and 12 protect the transport header, the main header, and its descriptor.

11. Protect all remaining transport header bits, beginning at the first bit of the protection/correction field, all main header bits, and all descriptor bits. Do not protect any payload bits. The protection is applied using 16 bytes on every group of 144 bytes. The total length of all fields being protected, including the error-code bits themselves, must be a multiple of 144 bytes. This is accomplished by the use of pad bits at the end of the transport header.

This implementation requires that memory be available to store the correction bytes for the entire length of transport header, main header, and its descriptor. For type 11, the total length of the protection code is 16 bytes for every 144. Thus, there are 16 extra protection bytes in each 144 bytes being stored, resulting in 128 bytes of data after decoding.

12. Same as 11, but with 16 bytes protecting every 80 bytes, resulting in 64 bytes of data after decoding. The total of all lengths, beginning at the first bit of the protection/correction field, must be a multiple of 80 bytes.

Types 13 and 14 protect the payload in addition to the transport header, the main header, and its descriptor.

13. Same as 11, except the payload is also protected.
14. Same as 12, except the payload is also protected.
15. Same as 11, except 4 bytes protect 32 bytes (total of 36 bytes for 32 bytes of data). This format is for short header formats, and provides no payload protection.

A.7.6.3 Interleaving

In addition to the above mechanisms for error protection, some of the error-protection formats can invoke interleaving. Interleaving can substantially reduce the problems associated with burst errors. Although the initial part of the transport header is subject to being "wiped out" by a burst error, presumably a copy of this section could be available previously in a redundancy transport header. Thus, once the protection format has been determined, then the rest of the transport header, beginning at the error-protection field, plus the main header, its descriptor, if present, and optionally the data payload, can all be protected from burst errors via interleaving in addition to error-correction methods.

Predefined interleaving methods can be incorporated with some of the types discussed above. Because interleaving is likely to involve as wide a spacing as is feasible, there will be a tradeoff between the length of the protected field, and natural multiples of the error-protection sizes. The error-protection group sizes for Hamming-type codes are much smaller than the error-protection group sizes for Reed-Solomon-type codes. Interleaving must be some multiple larger again. Thus, useful interleaving may be restricted to longer lengths of fields being protected. One possibility is to have the interleaving spacing be the error-protection group size divided into the total length.

However, this near-optimal format requires some complexity in unwinding the interleaving. For long protected fields, this may also involve a buffer that is the length of the field. Thus, there are potential issues to investigate with respect to how to universally and generally specify a powerful interleaving technique.

A.7.6.4 Error Detection

There is no provision for simple error detection in the above type examples. Such detection could be provided via cyclical

redundancy code (CRC), fire code, checksum, parity, or other check method. Such may be useful in some cases. However, the focus on error correction is based on the need for headers to be interpreted without error in order to serve their function.

The descriptor in the main header can be used for detection codes for data payloads that should be checked but need not be corrected. This need not be standard-specific, since the descriptor can be standard-independent. Thus, error detection, as opposed to correction, is more appropriate in the descriptor than in the transport header.

A.7.6.5 Parameters of Error Protection

The parameters of error detection shown in the above type examples need further investigation and refinement. The lengths and protection ratios proposed are known to be implementable in existing hardware and are expected to be convenient in practice. However, further investigation of optimal parameters for error protection may help refine or revise the parameters suggested above.

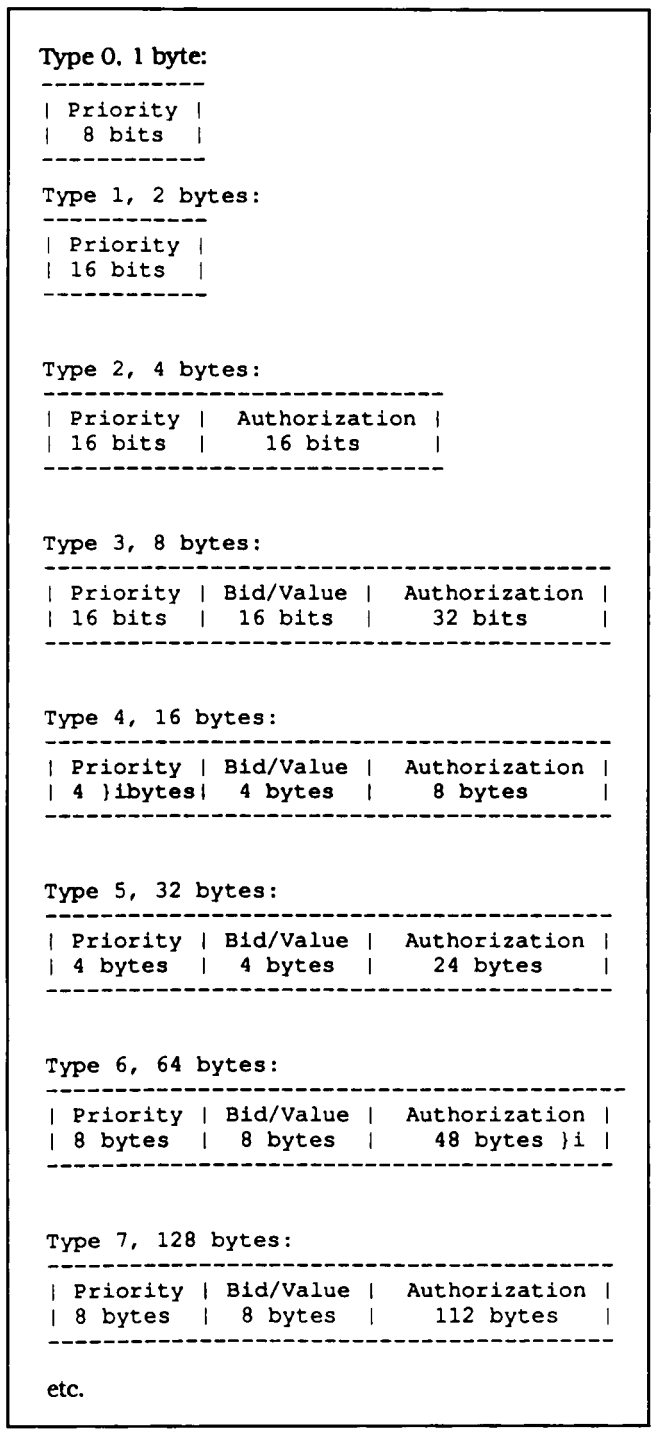

Figure A5. Field formats.

A.7.7 Priority and Authorization or Bid for Priority

Finite bandwidth resources, such as satellite channels, long fiber channels, long real-time computer channels, terrestrial broadcast channels, and other channels with long distances cause long latency, which naturally prevents error-retry. Thus, channel bandwidth allocation near saturation on real-time imagery streams takes the form of packet collisions. Such packets are most naturally the header/descriptor/payload combination, since each can have its own priority and each forms a constant priority grouping. The constant priority grouping would be the construction used by the originator.

When sharing a finite-bandwidth channel, it may be necessary to pass some data and drop other data. In order for the channel's controlling device to determine fairly which packets to pass and which to drop, priorities for packets might be provided. In many spatial-frequency-based compressed imagery formats such as DCT, subband, and wavelets, the high frequencies represent tiny picture detail that might be lost without much picture degradation. However, the spatial low frequencies, audio, and motion vectors must be heavily protected and may not be dropped without visible artifacts.

The type field would specify the length and format of the priority and/or authorization fields that follow. The length might have 2^type length (2 to the power of the value in the type field, being 2, 4, 8, 16, 32, 64, 128, etc., bytes). Type 0 still represents the absence of the priority field.

Since the priority and their authorization fields will compete at the highest level, it will be necessary for us to define their meaning at the outset. We will further need to define the mappings between the shorter and longer versions of each field.

The format of the fields might be as shown in Fig. A5. The priority field varies from 1 to 8 bytes, allowing very detailed priority levels.

The Bid/Value field allows a packet to have a "bidding price" in a collision with other packets. Such a bidding price would be a value if the price had previously been accepted. A value would imply that tossing the packet would break a contract for delivery to the packet. The meanings of the Bid/Value fields would be tied directly to authorization codes, which would indicate the following:

1. Whether the header was authorized to bid.
2. The "credit rating" (or importance) of the bidder. This could potentially weight the priority field.
3. Whether the bid had been previously accepted, so that the Bid/Value field was the value paid for the payload's delivery. In such a case, tossing the packet would violate the contract. Presumably such a case would occur only when more contracts had been made than were available, so that packets were only tossed by other similar accepted-bid packets with an established value. This is a similar problem to "overbooking" on airlines.
4. Authorization may affect the bid/value. If commissions are paid on some bids or values, and not on others, the net bid or value may differ. This is similar to the problem of bids in different currencies, or direct bookings versus using agents. Thus authorization can indicate the source and/or type of a bid for these purposes.

The priority should be registered in entirety. Thus, the meaning of priority codes might be defined by registration. However, it may be desirable to have priorities take simple linear precedence order, with higher values representing higher priorities. One possible solution is to define the first, or the first and second bytes of the priority to be linear magnitude precedence priority codes. Subsequent bytes, however, could be registered codes, with

unique meanings that are standardized to help resolve priority conflicts.

Other than the potential interactions of authorization on priority and bid/value, authorization can have the following very important uses.

A.7.8 The Authorization Field

Uses of authorization:

1. Pay-per-view target encryption codes (in lock step with receiving system).
2. Channel authorization. For example, is a satellite downlink channel signal authorized for use as a cable head-end?
3. Channel routing authorizations. For instance, are all authorized destinations only on network fork A, such that a source for subnetworks A and B need not carry the payload to subnetwork B. This is the function of supporting a subset of all of the outputs involved in a *Y* connection.
4. End-user authorization for teleconferencing, to indicate who can be included in the teleconference, including who may observe and who may originate.
5. Privacy and protection against unauthorized acceptance or origination of the signal in any use. For example, protection against real-time datastream hackers or unauthorized videophone wire tapping.
6. Authorized user enablement codes. Such codes would authorize user systems for future authorized codes. For example, when a cable subscriber adds a new channel, an enablement code would be sent to the decoder to add authorization interpretation and viewing for the new channel.
7. Diagnostic, statistic, and rating codes for exploring network loads, active users, show ratings, unintentional network disconnects, channel error rates, etc.
8. Copyright information indicating ownership.
9. Copyright fee structures, including where to pay fees.
10. Indications of who may edit a work, whether it may be included in other works, and fee structures for doing so.
11. Possibly an automated mechanism could be constructed to automatically negotiate rights based on prearranged "willing to pay" algorithms, so that clips can be included without undue complication.

A.7.9 Encryption

One function of the transport header is to provide one or more encryption keys for deciphering the payload and descriptor. Many protected users may wish to protect against unauthorized deciphering of the descriptor, since it may contain valuable information that could help in deciphering the payload. Codes could be used for encryption keys, for example, to unlock descriptors. Descriptors, in turn, may contain more elaborate encryption codes to further unlock the payload.

Based upon successful authorization code interactions, encryption codes can be deciphered and applied against the descriptor, the payload, or both. As usual, a type 0 represents that no encryption field is present. Fifteen types are available, with 15 prespecified associated lengths, for encryption. Although the lengths must be prespecified, the meaning of the encryption, or its associated technique, can be completely private. Complex encryption algorithms can be developed and updated between embedded codes in the receiving device, codes in the descriptor, and possibly algorithms transmitted and updated via descriptors.

A.7.10 Sequence Numbers

The sequence numbering field specifies not only packet ordering, but also windows of order and groupings. For example, in some systems various headers and their associated payloads form packets that can update the screen in any order during the frame time before the buffer switch for viewing. However, motion vectors might need to precede compressed image deltas. Thus not only packet grouping, but packet general ordering might be specified.

On some networks, lower-priority packets are delayed, rather than dropped. In such networks, it is necessary for the network controlling mechanisms to understand the bounds on acceptable delay for packets and groups of packets. This field contains a series of codes for defining tolerance of imagery and other real-time streams for being received out of order. Type 0 means no sequence field is present. Fifteen valid type fields with the associated prespecified lengths are available.

A.7.11 Timing Reconstruction

In real-time data streams, it is often necessary to reconstruct precise timing after this timing is disrupted during transport. Timing reconstruction information, concerning the times at which events should occur, are specified in this field. Times can be specified as absolute times, where the transport delays and their bounds are known. Relative times can be specified relative to an arbitrary "start of real-time stream" clock marker, which is set by the receiving device upon receiving the first displayable buffer load.

Synchronization between audio and image, between multiple audio streams, or between streams from multiple sources, is handled via the timing reconstruction field. Resynchronization for removing cumulative jitter effects can also be enabled through the use of this field.

A type of 0 indicates an absence of the timing reconstruction field. The 15 available codes will have prespecified lengths, although their timing meanings may be deferred from some of the types. Of course, each of the types can only receive a single meaning, which meaning must stay in place from then on. The lengths for such unspecified codes must all be specified in advance, however.

A.7.12 Reserved for the Future

This field is unspecified in content and length. Because the total length of the transport header is known, and because this is the last field in the header prior to the pad, this field {_ can maintain flexibility for future use by remaining completely unspecified. All other fields must at least have their lengths specified for each type value.

A.7.13 Pad Bits

Pad bits make the lengths simple for error-correction processing. This is accomplished by making the total of the error-correction/protection field range be the appropriate length for the error-correction format being used. For example, using a 64-bit length type, the length after the error-protection/correction field must be a multiple of 64 bits. Thus if the scope of the protection includes additional transport fields, such as priority and encryption, plus a header and its descriptor, the pad bits would make the sum a proper multiple of 64 bits.

Appendix B — Illustrative Examples of Header Decoding Using "C"

B.0 Background

It is often instructive to represent a design as a computer program written in some appropriate language (in this case C). It verifies the design and provides a basis for comparing the cost and performance of design alternatives. The C language was chosen because it is reasonably universal. Conciseness and consistency are foremost considerations in enabling comprehension and fair comparison; optimal performance is of secondary importance. Optimizations and enhancements would be added in preparation for commercial distribution.

Two programs are described. One decodes a compact header and one decodes an ASN.1 header. They are similar in appearance and use the same basic steps. The primary difference is the compact header decoder selects between multiple formats using table lookups, while an ASN.1 header has only one extensible

format. Each program extracts the block length and standard identifier from the header, and then calls a corresponding function to process the payload.

B.1 Compact Header Decoder

The following program decodes a packet with a compact header. If the packet format is predefined, it calls the corresponding predefined function, otherwise it extracts the standard identifier and block length in a manner similar to the algorithm described in Section 5.2.8. It uses the identifier to look up a decoding function (f), and ignores any blocks with unknown identifiers.

Two table lookups are used to decode the compact header. The length-type table (lt table) contains information used to decode predefined messages and block length. The identifier table (id table) contains information used to decode the standard identifier. One obvious optimization is to combine the two table lookups into a single 256-entry lookup. This reduces the instruction path for some cases, but increases memory requirements.

The identifier is left as a string of bytes used to compute a hash table lookup of a decoding function. If a sovereign state field exists, it is processed together with the standard identifier, but a separate hash table is used. The hash table lookup is performed by the procedure lookup0, which takes identifier address, identifier length, and table selection as arguments, and returns a pointer to the corresponding function (f).

B.1.1 Cautionary Notes

Certain header formats are not yet defined or are reserved for future use. The program below does not support these formats.

Predefined message types have not yet been standardized. To make the code complete a dummy function call, fake0, has been used. When the functions were standardized, the lt table would change accordingly.

Block length is assumed to fit within one 32-bit word. Extending the program and/or the C language to support larger word sizes, thus larger block lengths, is possible and likely to happen as 64-bit processor architectures emerge.

Bit-field ordering and assignment are not yet defined. Choices made in the program below will require further consideration in the context of standardization.

If an unknown identifier is encountered, the lookup function will return a pointer to an appropriate default function that ignores the payload and displays an informative message.

B.1.2 Program Text

The program has two parts — the first part contains table and procedure declarations, the second part (at the end) contains the dozen or so statements actually executed. Throughout the code descriptive notes (comments that are not executed) are placed between comment delimiters (/*...*/).

```
/*Compact header has one of two forms:
*
*Each character in the strings below represents a byte; bytes
*between square brackets are optional; payload bytes are not
*counted
*
*2-byte (minimum) for predefined messages:
*
* "ke[p...p]"
*
*Extended header for longer blocks:
*
* "kel[...l][e...e]i[...i]"
*
*Key:
*
* k :: key byte (length type and id type, presence of a readable
*description)
* e :: error byte
* l :: length byte
* i :: id byte
* p :: payload byte
*
*/
extern void fake0;      /*fake predefined func to init lt table*/
extern void null0;      /*null func for unused function entries*/

typedef struct {        /*lt table structure*/
  char length;          /*block length or length of length field*/
  void (*function)0;    /*predefined function for lt 0..5*/
  char id offset;       /*offset to ident field for lt 6..15*/
}Lt entry;

Lt entry lt table[16] = {                    /*lt table declaration*/
  {2, fake, 0}, {2, fake, 0}, {4, fake, 0}, {5, fake, 0},     /*0..3*/
  {6, fake, 0}, {0, fake, 0}, {1, null, 3}, {1, null, 4},     /*4..7*/
  {2, null, 4}, {2, null, 5}, {4, null, 6}, {4, null, 8},     /*8..11*/
  {6, null, 8}, {6, null, 10}, {0, null, 0}, {0, null, 0},    /*12..15*/
};

typedef struct{         /*id table structure*/
  char length;          /*length of ident field*/
  char table;           /*which table to use in lookup*/
}Id_entry;

Id_entry id_table[8]={                       /*id table declaration*/
  {1, 0}, {1, 1}, {2, 0}, {3, 0}, {2, 2}, {4, 2}, {8, 2}, {0, 0}
};

extern void (*lookup0)0;        /*id lookup function*/


/*EXECUTABLE CODE STARTS HERE*/

int decode compact header (pkt)   /*call standard decoder and
                                    return length*/
  unsigned char *pkt;             /*pointer to packet*/
{
  unsigned char *p=pkt;           /*working pointer to packet*/
  int lt;                         /*length type*/
  int length;                     /*block length*/
  Id entry *pid;                  /*id table pointer*/
  void (*f)0;                     /*standard function returned
                                    from lookup*/
  lt=*p>>4;                       /*get length type from bits
                                    4..7 of key*/
  if(lt<6)                        /*process predefined for-
                                    mats: lt<6*/
    {
      length=lt table[lt].        /*get block length from lt ta-
        length;                   ble*/
      (*lt table[lt].function)    /*call predefined function*/
        (p);
      return length;
    }

/*extract enough bytes to cover length field and shift out unused
bits*/

length=((int)*(p+2)<<24) | ((int)*(p+3)<<16) |
                                  ((int)*(p+4)<<8) | *(p+5);
length >>=32-(lt table[lt].length*8);

pid=&id table[*p & 0×7];        /*get id table ptr using bits 0..2
                                  of key*/
p+=lt table[lt].id offset;      /*move pointer to start of id
                                  field*/
f=lookup(p,pid->length,         /*lookup function*/
pid->table);
```

```
p+ =pid- >length                /*move pointer to start of pay-
                                 load*/
(*f)(p,length-(int)(p-pkt));    /*call function with payload
                                 ptr,length*/
return length;
}
```

B.2 ASN.1 Header Decoder

The following program decodes a packet with an ASN.1 header. The ASN.1 header is a compatible subset of the standard ASN.1 EXTERNAL type. As stated in Section 6.4, not all the flexibility of the standard EXTERNAL type is needed to meet the objectives. Thus this decoder has been specialized to support a level of function roughly comparable to that of the compact header decoder.

Decoding an ASN.1 header is performed by parsing a sequence of tokens. Block length is defined by one sequence, standard identification by a second. The header has short or extended forms. The short form is used for blocks of 128 bytes or less and is processed by extracting block length from a single byte. The extended form is processed by constructing block length from multiple bytes.

The identifier is left as a string of bytes used to compute a hash table lookup of a decoding function. The hash table lookup is performed by the procedure lookup0, which takes identifier address and length and returns a pointer to a corresponding function (*f*).

B.2.1 Cautionary Notes

Context-dependent headers are not decoded by the code below. They are decoded by standard decode procedures at a time they are expected. Block length is assumed to fit within one 32-bit word. Extending the program and/or the C language to support larger word sizes, thus larger block lengths, is possible and likely to happen as 64-bit processor architectures emerge. If an unknown identifier is encountered, the lookup function will return a pointer to an appropriate default function that ignores the payload and displays an informative message.

B.2.2 Program Text

The program has two parts – the first contains variable and procedure declarations, the second (at the end) contains the dozen or so statements actually executed. Throughout the code descriptive notes (comments that are not executed) are placed between comment delimiters (/*...*/).

```
/*ASN.1 header has one of three forms:
*
*Each character in the strings below represents a byte; bytes
*between square brackets are optional; payload bytes are not
*counted
*
*2-byte (minimum) header for context dependent messages:
*
* "tl[...l]"
*
*7-byte (minimum) header for short blocks:
*
* "tltli[...i]tl"
*
*Extended header for longer blocks:
*
* "tl[...l]tli[...i]tl[...l]"
*
*Key:
*
* t :: tag byte
* l :: length byte
* i :: id byte
*
*/

/* EXECUTABLE CODE STARTS HERE */

decode_asn1_header(pkt)         /*call standard decoder and
                                 return length*/
  unsigned char *pkt;           /*pointer to packet*/
{
extern void ((*lookup0))();     /*id lookup function*/
unsigned char *p=pkt;           /*working pointer to packet*/
unsigned int length;            /*block length*/
void (*f)0;                     /*standard function returned
                                 from lookup*/
unsigned int t;                 /*temp register*/

length=*(p+1);                  /*get 1st length byte*/
if (length<127)                 /*look for short form*/
 {
  t=*(p+3);                     /*get id field length*/
  f=lookup(p+4,t);              /*lookup function*/
  (*f)(p+t+6,length-
    (p+t+6-pkt));               /*call function*/
  return length+2;              /*return total length*/
 }
t=length-128;                   /*calc length of length field*/

/*extract enough bytes to cover length field and shift out unused
bits*/

length=((int)*(p+2)<<24) | ((int)*(p+3)<<16) |
                                ((int)*(p+4)<<8) | *(p+5);
length>>=32-(t*8);

p+ =t+3;                        /*move pointer to start of id
                                 field*/
f=lookup(p+1,*p):               /*lookup function*/
p+ =*p+t+3;                     /*move pointer to start of
                                 payload*/
(*f)(p,length-(int)(p-pkt));    /*call func (payload ptr and
                                 size)*/
return length+t+2;              /*return total length*/
}
```

Appendix C – Header/Descriptor Task Force Members

Will Stackhouse, Chairman	Jet Propulsion Laboratory
David H. Staelin, Vice-Chairman	Massachusetts Institute of Technology
Walter Bender	Massachusetts Institute of Technology
Rita Brennan	Apple Computer, Inc.
Wayne E. Bretl	Zenith Electronics
David C. Carver*	Digital Equipment Corp.
Gary Demos	Demografx
Ephraim Feig	IBM
Branko Gerovac*	Digital Equipment Corp.
Jeffrey W. Johnston	Eastman Kodak Co.
Michael Liebhold	Apple Computer, Inc.
Lee McKnight	Massachusetts Institute of Technology
Arun Netravali	AT&T Bell Laboratories
Bruce Sidran	Bellcore
D. Scott Silver	Tektronix, Inc.
Richard J. Solomon	Massachusetts Institute of Technology
David L. Tennenhouse	Massachusetts Institute of Technology
David L. Trzcinski	PictureTel
Ken C. Yang	Ampex Corp.

*Associate members making significant written contributions.