NOAA GOES LRIT Mission Specific Data

Introduction

Specifications for the reception and decoding of the GOES Low-Rate Information Transmission (LRIT) are set forth in the NOAA LRIT Receiver Specification (System Specifications). The specification contains a GOES mission specific implementation of the CGMS 03 LRIT/HRIT Global Specification, Issue 2.6, August 12, 1999. While the GOES LRIT specification seeks to provide developers with the mission specific implementation of the CGMS 03 specification, it does not include details on some key aspects of the GOES LRIT transmission. In particular, the CGMS recommendation for image compression is JPEG whereas GOES LRIT uses lossless Rice compression on image files. This does not violate the CGMS specification since JPEG is only a recommendation and other types of compression are allowed. Further issues for developers of GOES LRIT software can arise in the interpretation of the CGMS specification vs the GOES LRIT implementation in the transport layer protocol. The Rice compression algorithm used in GOES LRIT is licensed and not generally available to terminal developers. However, software has been made available on the NOAA GOES LRIT Web site to allow developers to perform the image decompression. It is the intent of this paper to present a simplified view of the steps required to decode the LRIT data with special emphasis on implementing Rice decompression.

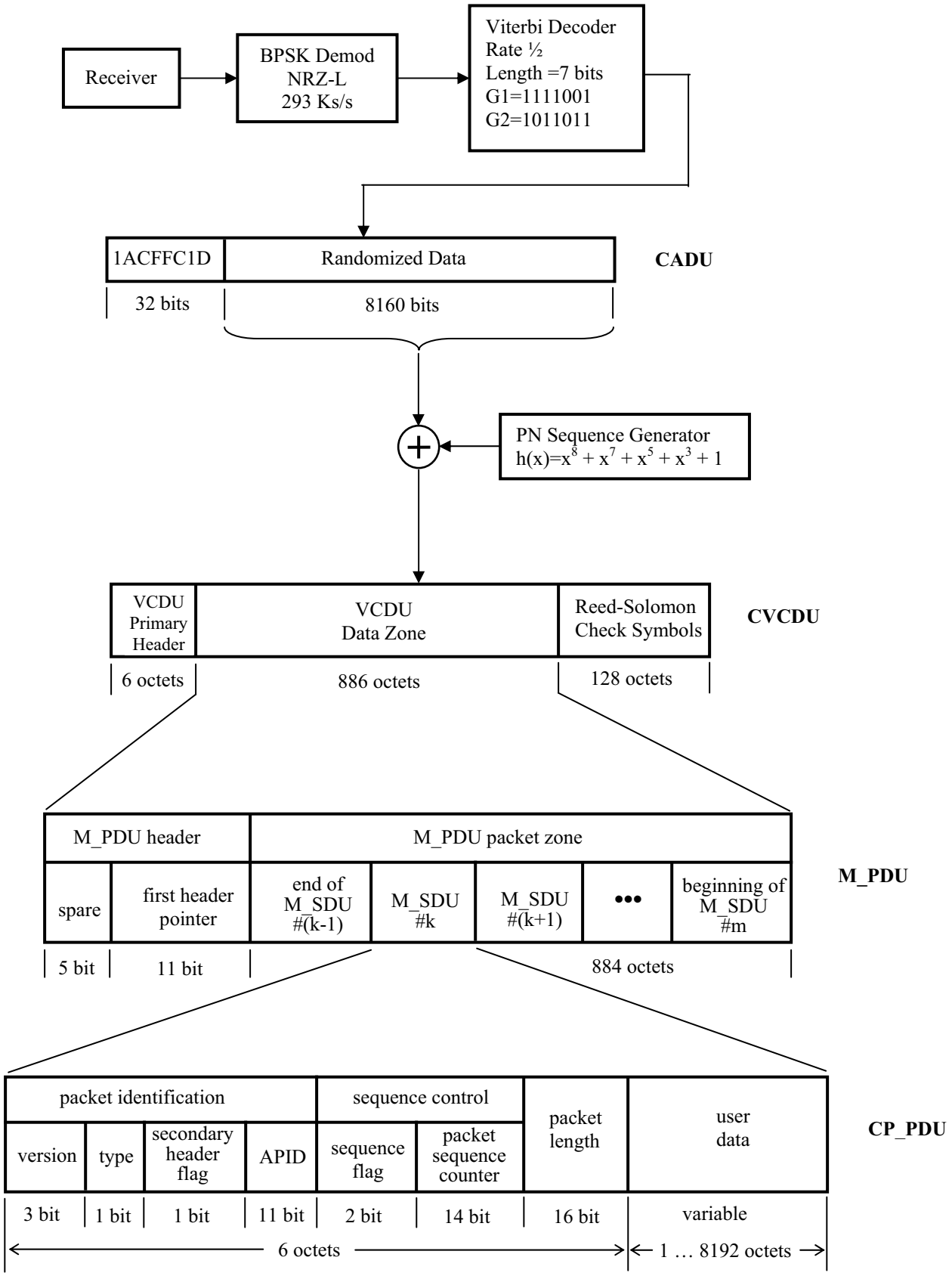
Data Processing

The processes required to obtain an LRIT file from the transmitted data are outlined in Figures 1 and 3. The physical layer shown in Figure 1 includes a receiver to downconvert the 1691 MHz received signal to an appropriate IF frequency followed by a BPSK demodulator to obtain the NRZ-L coded data stream at 293 Ks/s. The convolutionally encoded data is passed through a rate ½ Viterbi decoder with constraint length=7, G1=1111001, and G2=1011011 to obtain the final data stream to be decoded. The data is received in blocks of 8160 bits with each block preceded by a 32 bit pattern (1ACFFC1DH) for block synchronization. The entire 8192 bit block is the "channel access data unit" (**CADU**) in the CGMS specification. The 8160 bits of data have been randomized to assure sufficient data transitions for the demodulation process. Thus the first step in decoding the block is to perform data de-randomization. This is accomplished by bitwise exclusive or-ing the 8160 bits of data with a PN sequence whose generator polynomial is given by

$$h(x)=x^8 + x^7 + x^5 + x^3 +1.$$

A sample program for generating the PN sequence and performing the de-randomization is given in Appendix A. The resulting 8160 bit block is the "coded virtual channel data

unit" or **CVCDU**. The **CVCDU** is made up of a 6 octet VCDU primary header, 886 octets of VCDU data, followed by 128 octets of Reed-Solomon check symbols. The



Reed-Solomon code (255,223) is calculated over the entire 892 octets of VCDU primary header and VCDU data. Reed-Solomon coding does not actually alter the original data, thus, under high signal to noise conditions, the VCDU header and data could be used without R-S decoding. However, to assure minimal data errors and determine the integrity of the data, it is recommended that the R-S decoding step be performed. Sample code for implementing the function is given in Appendix A.

The VCDU primary header is shown in Figure 2. It contains information including spacecraft identifier, version number, a sequential VCDU counter (modulo 16777216), and a virtual channel identifier (VC ID). If the VC ID is set to 63 then the VCDU is a fill VCDU and contains no user data. The block is discarded and a new **CADU** is received for processing.

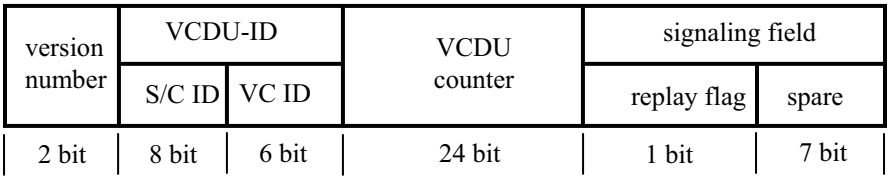


Figure 2. VCDU Primary Header

The 886 octets in the VCDU contain the "multiplexing protocol data units" (**M_PDU**) as shown in Figure 1. The **M_PDU** contains one or more "multiplexing service data units" (M_SDU) each of which contains a packet or portion of a packet carrying the user data. The first 16 bits of the **M_PDU** contain a "first header pointer" which identifies the offset (in octets) from the beginning of the **M_PDU** packet zone to the first M_SDU that contains a packet with a header (e.g. M_SDU #k in Figure 1.) All data prior to this packet (if any) is the last portion of data from an M_SDU in a preceding block. M_SDU packets can span over multiple **M_PDU** blocks. If the entire **M_PDU** packet zone contains data from a previous block (i.e. **M_PDU** packet zone does not contain a header), then the "first header pointer" will be set to 2047.

The M_SDU packets contain the fundamental user data packets identified as "CCSDS path protocol data units" (**CP_PDU**) in CGMS 03. The structure of these variable length packets is shown in Figure 1 and repeated in Figure 3. LRIT files are assigned an "application process identifier" (APID) based on their priority and then broken up into variable length segments for transmission. Each segment, which can be from 1 to 8190

octets in length, receives a 16 bit CRC error control field at its end resulting in the 1 to 8192 octet user data in the **CP_PDU** structure shown in Figures 1 and 3. The user data is concatenated to a 6 octet header containing information about the user data. The header contains the APID identifier, a sequence flag that identifies whether the packet contains the first, middle, or last segment of the user data, a packet sequence counter that

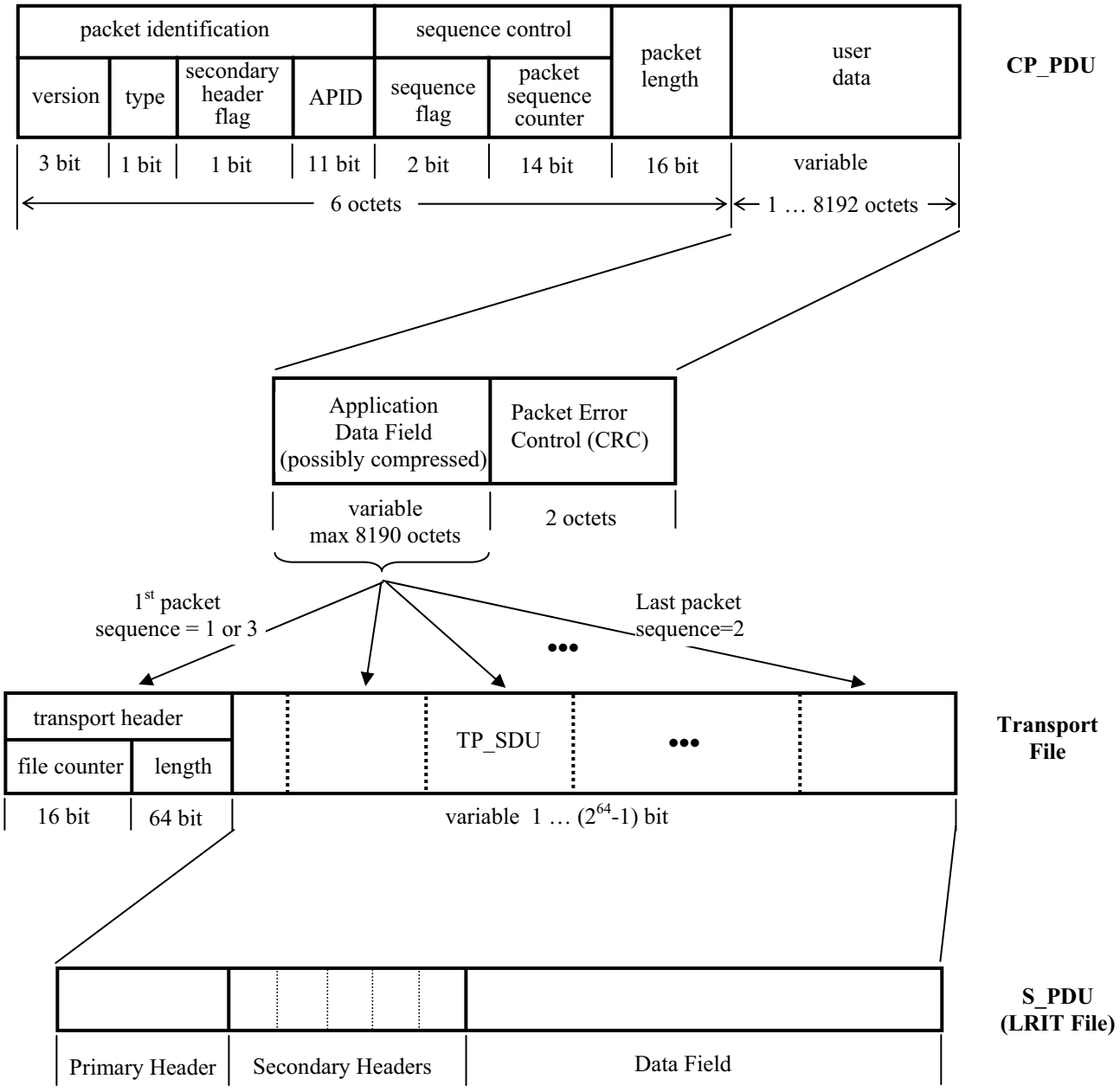


Figure 3.

increments modulo 16384 for the specified APID, and a packet length field identifying the number of octets in the user data field minus 1. The CRC checksum is computed over the variable length "Application Data Field" portion of the **CP_PDU** user data. The generator polynomial for the CRC is given by

$$g(x) = x^{16} + x^{12} + x^5 + 1$$

The function is initialized to "all ones" prior to CRC calculation for each segment. A coding sample of the CRC calculation is provided in Appendix A.

As the **CP_PDU** packets arrive for a given APID, the contents of the "Application Data Field" are concatenated together under control of the sequence flag to form the **Transport File** as defined in CGMS 03. The transport file starts with an 80 bit transport header followed by a variable length Transport Service Data Unit (**TP_SDU**). The **TP_SDU** in turn contains a Service Protocol Data Unit (**S_PDU**) which is the desired LRIT file associated with the specified APID.

The CGMS 03 specification is somewhat ambiguous in the description of the transport file structure. Section 6.2.2 states that the transport file shall be split into blocks of 8190 octets in size with each block receiving a CRC checksum field. However, in the same paragraph, the statement is made that this results in segments of up to 8192 octets in size. The NOAA LRIT implementation splits the transport file into variable length segments with each segment receiving the CRC checksum. These segments then become the user data portion of the **CP_PDU** packet.

The 80 bit transport header consists of a 16 bit sequential counter (modulo 2^{16}) incremented with each received transport file and a 64 bit length field indicating the number of bits in the **TP_SDU** data field. The length field indicates the number of bits prior to any data compression. Data compression in the GOES LRIT implementation occurs at the **CP_PDU** packet level.

Referring to Figure 3, if the sequence flag in the **CP_PDU** packet is a "3", then the user data contains a complete transport file starting with the 80 bit header followed by a variable length LRIT file. The LRIT file will contain the mandatory **primary header record** (header type 0) followed by one or more **secondary header records** in turn followed by application data (if any.) The structure and meaning of these header records are described in detail in th LRIT specification(s). If the sequence flag is a "1", then the user data will contain the transport header and the first portion of an LRIT file which generally will include the **primary and secondary header records.** Subsequent **CP_PDU** packets will have a sequence flag equal to "0" indicating the user data is a continuation of the transport file data contents up until the last segment of the file which will be designated with a sequence flag equal to "2".

If the transmitted data is not compressed, then the above procedure will lead to a complete LRIT file in the **TP_SDU** contents of the transport file. If the transmitted file contains a Rice compressed image, then decompression of the **CP_PDU** user data must occur prior to concatenation in the transport file. To determine if the data is compressed, the first packet(s) must be decoded to obtain the **primary** and **secondary** LRIT header records. Rice compression is only used on image files, thus, the third field in the LRIT **primary header** (i.e. File type code) shown in Figure 4 must be set to a "0" designating the file is an "image data file" as specified in Table 4-2 of the CGMS 03 specification. If the file is an image data file, then the LRIT **primary header** will be followed with a mandatory **Image Structure Record** (Header Type 1) in the **secondary headers.** The **Image Structure Record** is reproduced in Figure 5.

size in octets	data type	contents
1	integer, unsigned	header type, set to **0**
2	integer, unsigned	header record length, set to 16
1	integer, unsigned	file type code, determining the top level structure of the file data field
4	integer, unsigned	total header length, specifying the total size of all header records (including this one) in octets
8	integer, unsigned	data field length, specifying the total size of the file data field in bits

Figure 4. Primary Header Record

size in octets	data type	contents	abbreviation
1	integer, unsigned	header type, set to **1**	
2	integer, unsigned	header record length, set to 9	
1	integer, unsigned	number of bits per pixel (1 … 255)	NB
2	integer, unsigned	number of columns (1 … 65535)	NC
2	integer, unsigned	number of lines (1 … 65535)	NL
1	integer, unsigned	compression flag (0,1,2)	CFLG

Figure 5. Image Structure Record

If the image data is Rice compressed, then the compression flag, CFLG in the Image Structure Record will be set to a "1". For uncompressed data, this value will be "0." If the data is Rice compressed, then a third secondary header must be decoded to obtain the Rice compression parameters. This is the NOAA specific **Rice Compression Record** with a header type equal to "131" as shown in Figure 6.

Field Name	size	Type	Description	Value
headerType	1	Uint	NOAA-specific header type code for Rice Compression Header	131
headerLength	2	Uint	Header length in octets	7
flags	2	Uint	Compression option flags	Sum (bit-wise "or") of values in Table 19
pixelsPerBlock	1	Uint	Number of pixels in each CDS	Even number, $4 \leq value \leq 64$
scanLinesPerPacket	1	Uint	Number of compressed scan lines in one packet	1-255

Figure 6. Rice Compression Record

If the **Rice Compression Record** is missing from the secondary headers, then the following default values are to be used in the Rice decompression algorithm.

Field	Default Value
flags	49
pixelsPerBlock	16
scanLinesPerPacket	1

At the outset it was stated that the Rice compression algorithm used to compress NOAA LRIT images is licensed and not readily available to developers of LRIT software. To circumvent this, the "Domain 6 LRIT Reception Software Beta version" posted on the User Station Software web site includes software that can be used in user code to perform the decompression. In particular, the class "CriceDecompression" located in the header

file "PacketDecompression.h" in the d6/LritRice folder provides a simple means for decompressing the data. To use the class, the file "LritRice.lib" located in d6/Common must be included in the build.

The class constructor requires the passing of five parameters for proper initialization as defined in the following:

```
CRiceDecompression(
                int Mask,
                int BitsPerPixel,
                int PixelsPerBlock,
                int PixelsPerScanline,
                int ScanLinesPerPacket);
```

The "Mask" parameter is the *flags* field in the **Rice Compression Record** shown in Figure 6. It will typically be the default value of 49 (31H). *BitsPerPixel* is obtained from the third field (NB) in the **Image Structure Record** of Figure 5. The *PixelsPerBlock* and *ScanLinesPerPacket* parameters are the fourth and fifth fields in the **Rice Compression Record**, respectively. The *PixelsPerScanline* parameter is the number of columns (NC) entry in the **Image Structure Record** of Figure 5.

To implement the decompression, an instance of the class is first created with the above parameters passed as arguments.

```
CriceDecompression  *Rice = new CriceDecompression(flags, NB, PPB, NC, SLPP);
```

Recall that the compressed image data is in the "Application Data Field" portion of the **CP_PDU** packet user data field shown in Figure 2. To decompress the data, the member function

```
Rice->Decompress(dataptr, datalength)
```

is called where *dataptr* is a pointer to the beginning of the **CP_PDU** user data field and *datalength* is the number of compressed bytes in the field. The **CP_PDU** "packet length*"* field indicates the total number of bytes in the user data minus 1. This includes the two CRC checksum bytes. Thus, the correct value for *datalength* in the Decompress function is simply *datalength* = packet length – 1.

The Rice->Decompress function will return a value of TRUE if it succeeds and FALSE if an error occurs. If the operation is successful, then two other member functions are called to retrieve the decompressed data. The function

Rice->Ptr( );

returns a pointer of type (unsigned char*) to the decompressed data and the function

Rice->Size( );

returns an (unsigned long) count of the number of bytes in the decompressed data. Thus, for each **CP_PDU** packet containing Rice compressed data, a suitable routine for decompression would be

```
Rice->Decompress(dataptr, datalength);
memcpy(tppointer, Rice->Ptr(), Rice->Size());
tppointer + = Rice->Size;
```

where *tppointer* is a pointer to the location in the transport file where the decompressed data is to be concatenated. When the last **CP_PDU** packet (sequence = 2) is decompressed, the **TP_SDU** contents of the transport file will contain the completed, decompressed LRIT image file. Display of the image (or image segment) follows the specifications set forth in the LRIT Receiver Specification or CGMS 03. The image structure including the number of bits per pixel (NB), number of columns in the image (NC), and the number of lines in the image (NL) are found in the **Image Structure Record** shown in Figure 5.

Conclusion

A simplified description of the steps required to decode the NOAA GOES LRIT mission specific data transmissions into LRIT application files has been presented. The intent is to provide terminal software developers a visualization of the data flow as it progresses through the various layers defined by the LRIT system specification(s). NOAA has posted beta software (Domain 6) on the LRIT Web site that can be used in its entirety to implement processing and display of received LRIT files. For developers who choose to write their own receiver specific code, functions are available in the Domain 6 software to allow implementation of NOAA LRIT specific processes such as Rice decompression of image files. Examples of the usage of this code are included in the main body of this

paper with more specific examples in the Appendix. It is important to note that the data flow diagrams (Figures 1 and 3) are simplified examples and the reader must refer to the NOAA LRIT Receiver Specification and CGMS 03 HRIT/LRIT Global Specification for the specifics on processing the various data layers.

Appendix A

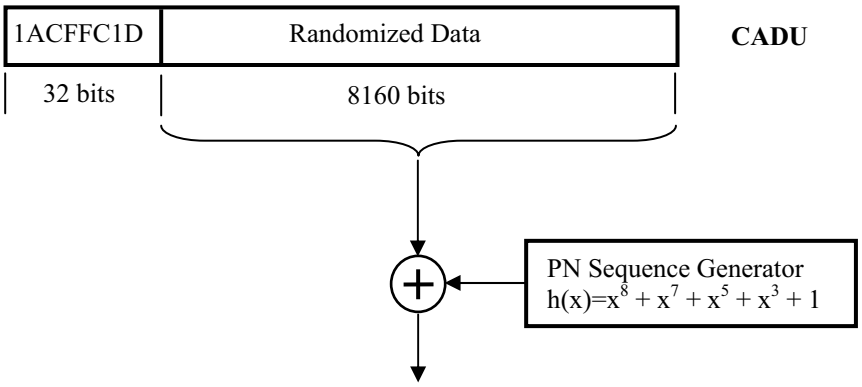
Sample Code Implementations

De-Randomization

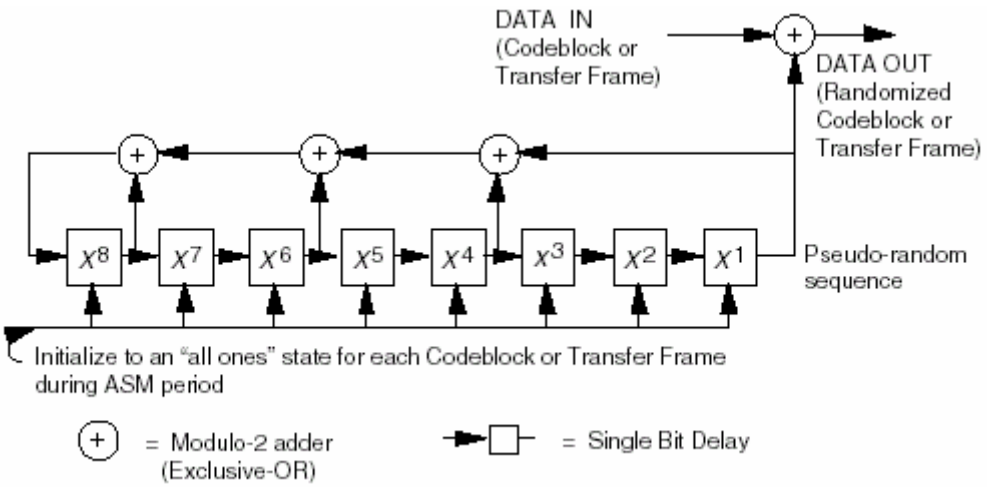
Data is received out of the physical layer as blocks of length 8192 bits. Each block starts with a 32 bit sync pattern with the remaining 8160 bits or 1020 octets representing the user data. To assure sufficient data transitions in the demodulation process, the data has been randomized by bitwise exclusive or-ing it with a pseudorandom (PN) pattern generated using the polynomial

$$h(x)=x^8 + x^7 + x^5 + x^3 +1.$$

De-randomizing the data requires that it again be exclusive or-ed with an identical PN sequence.



The CCSDS 101.0-B-6 recommendation for "Channel Telemetry Coding" suggests a possible implementation of the PN sequence generator described by the following diagram.



Direct implementation of the diagram in code implies bit-by-bit de-randomization of the input data stream. For many applications, it is more efficient to de-randomize the data at a byte level. The following code pre-generates an array of PN characters that are then sequentially byte-wise exclusive or-ed with the incoming data to perform the de-randomization.

```
unsigned char pn[1020];
int i,j;
unsigned char fdbk ,randm=0xff;

for(i=0;i<1020;i++)
{
  pn[i]=0;
  for(j=0;j<=7;j++)
  {
    pn[i]=pn[i]<<1;
    if(randm&0x80) pn[i]++;
    fdbk=randm&0x95;
    randm=randm<<1;
    if((((fdbk&0x80)^(0x80&fdbk<<3))^(0x80&(fdbk<<5)))^(0x80&(fdbk<<7)))randm++;
  }
}
```

The 1020 byte table produced by this code is a repeated sequence of the 255 bytes generated by the randomizing polynomial.

pn[] =

```
FF 48 0E C0 9A 0D 70 BC 8E 2C 93 AD A7 B7 46 CE
5A 97 7D CC 32 A2 BF 3E 0A 10 F1 88 94 CD EA B1
FE 90 1D 81 34 1A E1 79 1C 59 27 5B 4F 6E 8D 9C
B5 2E FB 98 65 45 7E 7C 14 21 E3 11 29 9B D5 63
FD 20 3B 02 68 35 C2 F2 38 B2 4E B6 9E DD 1B 39
6A 5D F7 30 CA 8A FC F8 28 43 C6 22 53 37 AA C7
FA 40 76 04 D0 6B 85 E4 71 64 9D 6D 3D BA 36 72
D4 BB EE 61 95 15 F9 F0 50 87 8C 44 A6 6F 55 8F
F4 80 EC 09 A0 D7 0B C8 E2 C9 3A DA 7B 74 6C E5
A9 77 DC C3 2A 2B F3 E0 A1 0F 18 89 4C DE AB 1F
E9 01 D8 13 41 AE 17 91 C5 92 75 B4 F6 E8 D9 CB
52 EF B9 86 54 57 E7 C1 42 1E 31 12 99 BD 56 3F
D2 03 B0 26 83 5C 2F 23 8B 24 EB 69 ED D1 B3 96
A5 DF 73 0C A8 AF CF 82 84 3C 62 25 33 7A AC 7F
A4 07 60 4D 06 B8 5E 47 16 49 D6 D3 DB A3 67 2D
4B BE E6 19 51 5F 9F 05 08 78 C4 4A 66 F5 58
                    •
                    •
                    •
```

De-randomizing each received **CADU** can then be accomplished with the following routine

```
int i;
PUCHAR bufptr;

for(i=0;i<1020;i++)
  {
      *bufptr=pn[i]^*bufptr;
      bufptr++;
  }
```

where *bufptr* is a pointer initially set to the beginning of the **CADU** randomized data.

Reed-Solomon Decoding

The 1020 octet **CVCDU** block that results after data de-randomization includes a 6 octet VCDU header, 886 octets of VCDU data, and a 128 octet field of Reed-Solomon check symbols. The (255,223) Reed-Solomon code is calculated over the entire 892 octets of VCDU primary header and VCDU data. The Domain 6 software on the LRIT Web site includes a class than can be used to simply implement the Reed-Solomon decoding. The class is included in the files "Reed Solomon.cpp" and "Reed Solomon.hpp" located in the d6/Common folder. Creating an instance of the class requires the passing of eight parameters defined by

CReedSolomon(int BitsPerSymbol, int CorrectableErrors, int mo, int poa,
int VirtualFill, int Interleave, int FrameSyncLength, int mode);

Definitions of the parameters can be found in the referenced files, however, for the GOES LRIT transmission, the proper values are:

CReedSolomon(8,16,112,11,0,4,0,1)

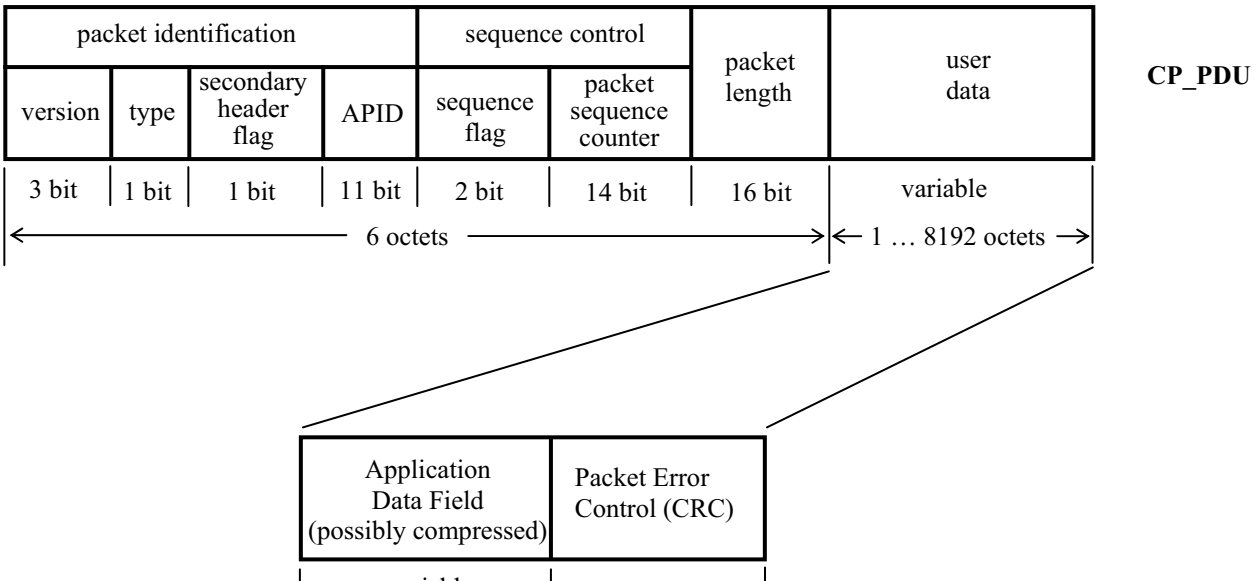
By including the file "Reed Solomon.cpp" in the build and the file "Reed Solomon.hpp" in the header files, the following code can be used to provide data correction.

```
//  Header files

#include "Reed Solomon.hpp"



// Global Variables

CReedSolomon LRITrs(8,16,112,11,0,4,0,1);
PUCHAR cvcduptr;

// Main Body

        •
        •
        •

LRITrs.Decode((unsigned char *) cvcduptr);
```

In the above code, *cvcduptr* is a pointer to the beginning of the 1020 octet **CVCDU** block. Calling the member function "LRITrs.Decode" performs the error correction on the block.

CRC Error Check

The variable length user data portion of the **CP_PDU** packet includes a 2 octet CRC checksum field.



The checksum is calculated over the Application Data Field portion of the packet using the generator polynomial

$$g(x) = x^{16} + x^{12} + x^5 + 1$$

The function is initialized to "all ones" prior to CRC calculation for each segment. Many possible implementations of CRC decoding algorithms have been published in the literature. The technique shown in the following code uses table look-up to speed up the process. In the code, *dataptr* is a pointer to the beginning of the **CP_PDU** user data field and *datalength* is the number of bytes in the Application Data Field. The **CP_PDU** "packet length" field indicates the total number of bytes in the user data minus 1. This includes the two CRC checksum bytes. Thus, the correct value for *datalength* in the call to the CRC function is simply *datalength* = packet length – 1.

```
//Global Variables

unsigned short crcTable[] = {
   0x0000, 0x1021, 0x2042, 0x3063, 0x4084, 0x50A5, 0x60C6, 0x70E7,
   0x8108, 0x9129, 0xA14A, 0xB16B, 0xC18C, 0xD1AD, 0xE1CE, 0xF1EF,
   0x1231, 0x0210, 0x3273, 0x2252, 0x52B5, 0x4294, 0x72F7, 0x62D6,
   0x9339, 0x8318, 0xB37B, 0xA35A, 0xD3BD, 0xC39C, 0xF3FF, 0xE3DE,
   0x2462, 0x3443, 0x0420, 0x1401, 0x64E6, 0x74C7, 0x44A4, 0x5485,
   0xA56A, 0xB54B, 0x8528, 0x9509, 0xE5EE, 0xF5CF, 0xC5AC, 0xD58D,
   0x3653, 0x2672, 0x1611, 0x0630, 0x76D7, 0x66F6, 0x5695, 0x46B4,
   0xB75B, 0xA77A, 0x9719, 0x8738, 0xF7DF, 0xE7FE, 0xD79D, 0xC7BC,
   0x48C4, 0x58E5, 0x6886, 0x78A7, 0x0840, 0x1861, 0x2802, 0x3823,
   0xC9CC, 0xD9ED, 0xE98E, 0xF9AF, 0x8948, 0x9969, 0xA90A, 0xB92B,
   0x5AF5, 0x4AD4, 0x7AB7, 0x6A96, 0x1A71, 0x0A50, 0x3A33, 0x2A12,
   0xDBFD, 0xCBDC, 0xFBBF, 0xEB9E, 0x9B79, 0x8B58, 0xBB3B, 0xAB1A,
   0x6CA6, 0x7C87, 0x4CE4, 0x5CC5, 0x2C22, 0x3C03, 0x0C60, 0x1C41,
   0xEDAE, 0xFD8F, 0xCDEC, 0xDDCD, 0xAD2A, 0xBD0B, 0x8dD68, 0x9D49,
   0x7E97, 0x6EB6, 0x5ED5, 0x4EF4, 0x3E13, 0x2E32, 0x1E51, 0x0E70,
   0xFF9F, 0xEFBE, 0xDFDD, 0xCFFC, 0xBF1B, 0xAF3A, 0x9F59, 0x8F78,
   0x9188, 0x81A9, 0xB1CA, 0xA1EB, 0xD10C, 0xC12D, 0xF14E, 0xE16F,
   0x1080, 0x00A1, 0x30C2, 0x20E3, 0x5004, 0x4025, 0x7046, 0x6067,
   0x83B9, 0x9398, 0xA3FB, 0xB3DA, 0xC33D, 0xD31C, 0xE37F, 0xF35E,
   0x02B1, 0x1290, 0x22F3, 0x32D2, 0x4235, 0x5214, 0x6277, 0x7256,
   0xB5EA, 0xA5CB, 0x95A8, 0x8589, 0xF56E, 0xE54F, 0xD52C, 0xC50D,
   0x34E2, 0x24C3, 0x14A0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
   0xA7DB, 0xB7FA, 0x8799, 0x97B8, 0xE75F, 0xF77E, 0xC71D, 0xD73C,
   0x26D3, 0x36F2, 0x0691, 0x16B0, 0x6657, 0x7676, 0x4615, 0x5634,
   0xD94C, 0xC96D, 0xF90E, 0xE92F, 0x99C8, 0x89E9, 0xB98A, 0xA9AB,
```

```
    0x5844, 0x4865, 0x7806, 0x6827, 0x18C0, 0x08E1, 0x3882, 0x28A3,
    0xCB7D, 0xDB5C, 0xEB3F, 0xFB1E, 0x8BF9, 0x9BD8, 0xABBB, 0xBB9A,
    0x4A75, 0x5A54, 0x6A37, 0x7A16, 0x0AF1, 0x1AD0, 0x2AB3, 0x3A92,
    0xFD2E, 0xED0F, 0xDD6C, 0xCD4D, 0xBDAA, 0xAD8B, 0x9DE8, 0x8DC9,
    0x7C26, 0x6C07, 0x5C64, 0x4C45, 0x3CA2, 0x2C83, 0x1CE0, 0x0CC1,
    0xEF1F, 0xFF3E, 0xCF5D, 0xDF7C, 0xAF9B, 0xBFBA, 0x8FD9, 0x9FF8,
    0x6E17, 0x7E36, 0x4E55, 0x5E74, 0x2E93, 0x3EB2, 0x0ED1, 0x1EF0,
};


//Forward declaration of functions

unsigned short CRC(const unsigned char* , int );


//Main Body

      •
      •
      •

unsigned short CRC(const unsigned char* buf, int length)
{
     unsigned short Crc=0xffff,ind=0;
     while(length--)
        {
            Crc=(Crc<<8)^crcTable[(Crc>>8)^(unsigned short)buf[ind++]];
        }
     return Crc;
}

      •
      •
      •
//CRC Calculation Routine

unsigned short CRCchecksum;

CRCchecksum=CRC(dataptr,datalength);
```

After the call to the CRC function, the variable *CRCchecksum* can be compared to the 2 octet packet error control field in the **CP_PDU** packet to determine the presence of any packet errors.

Rice Decompression

US Government Disclaimer

The United States Government nor any of its data or content provider shall be liable for any errors in the content of this document, or for any actions taken in reliance thereon. All data and information contained herein is provided for informational purposes only, and is the users responsibility if he/she decides to use or incorporate into their design.

The following code sample is an implementation of the Rice decompression outlined in detail in the main text of this document. It is assumed that the LRIT **primary** and **secondary** headers have been decoded and the necessary parameters extracted to pass to the CRiceDecompression functions. The "CriceDecompression" class is located in the header file "PacketDecompression.h" in the d6/LritRice folder. The "LritRice.lib" file located in d6/Common must be included in the build.

```
//Header Files

#include "PacketDecompression.h"

//Global Variables

int flags,NB,PPB,NC,SLPP;


//Main Body

      •
      •
      •
// Decompression routine.  dataptr is a pointer to the beginning of the
CP_PDU packet user data field.  datalength is the length of the
compressed Application Data Field portion of the packet (user data less
the CRC checksum bytes). datalength is equal to the packet length field
minus 1.  tppointer is a pointer to the insertion point of the
decompressed data into the transport file.

CriceDecompression  *Rice = new CriceDecompression(flags, NB, PPB, NC, SLPP);
if(!Rice->Decompress(dataptr, datalength))
      {
            //Error handling routine
      }
memcpy(tppointer, Rice->Ptr(), Rice->Size());
tppointer+=Rice->Size;
```