Proceedings of the 4th WSEAS/IASME Int. Conf. on System Science and Simulation in Engineering, Tenerife, Spain, December 16-18, 2005 (pp177-185)

Interpreting an Arbitrary Data Stream

LIHUA WANG AND LUIZ F. CAPRETZ
Department of Electrical and Computer Engineering
University of Western Ontario
London, Ontario, N6A5B9
CANADA

Abstract: This paper describes a new method that can interpret and extract information from a machine-level binary data stream. This method has two major components: one is a Data Format Scripting Language (DFSL); the other is generic Data Parsing Application Software. The DFSL is a scripting language that specifies the physical layout and semantic constraints for an associated binary data stream. This language can be used to parse a data stream with a specified data format and convert the data into meaningful data structure with text notation. The Data Parsing Application Software is a generic program used to parse the script and its associated raw data. Depending on the script, this program can interpret the meaning of individual data fields and their values. This language and the developed system can be used in the electrical engineering area, especially where the demand for manipulation of real machine-level binary digits is high.

Key-Words: Data Stream, Table Scripting Languages, Data Format, Data Description Languages

1 Introduction

Information extraction from an arbitrary data stream is an important task in the developing and debugging processes in electrical engineering, such as hardware device development, network protocol processing, and protocol testing and developing. The raw data can be binary data or one of the standard text schemes (Unicode, ASCII, EBCDIC, etc); however any data can be regarded as a sequence of bits.

Data format is the key of determining the physical layout and semantic meanings of the data. Thus understanding the data format and producing a parser are crucial steps of extracting information and for future manipulation [1]. Understanding the complicated data format is not always easy. Current data format documents are typically described in natural human language, which is prone to ambiguities and misunderstanding. How to specify a data format accurately is an interesting research task but is beyond the scope of this paper. Here, we assume our users are professional engineers who have sufficient information to help them in their understanding of the specific data format.

The question then becomes how to produce the parser efficiently. Currently, a programmer must choose a language, convert the documented data format description into data structures such as C structures; then the data must be read into memory, some operations performed, and the data written back to external storage [2].

The C programming language is the most commonly chosen language for writing such programs, especially to implement real-time algorithms in a system that interfaces with bare hardware devices [3]. Engineers incur time-consuming and error-prone penalties from low efficiency C languages when extracting information from raw data.

Our solution is to create a scripting language that is easy to learn, straightforward to understand, and agile to fulfil a user's requirements at the same time. We have also developed a generic application to parse the script and the raw data.

The advantages of this solution are:

1. The only tool a user needs is a text editor; no compiler/linker or software development system is necessary.
2. Users don't need to learn programming.
3. The language is easy to use with simple

syntax and rich semantic meaning. It is very close to the data format configuration table used to define the data format.

4. It saves developing time.
5. Many functions are optimised and hidden from the user such as file access and data manipulation.

The user can write a script for a specific data format using this language and the associated application can parse the script and the raw data. The output is a meaningful interpretation of the data fields. This process takes shorter amount of time than using functional language, such as C.

2 Motivation

The original idea for this project came from a problem faced by many engineers in hardware device development. That is the need for an easier way to parse and interpret the arbitrary bit streams.

Most devices such as drivers and interface cards have control and status registers. The device is controlled by setting and clearing particular bits in the *control register* (see Figure 1), while its status is obtained by examining bits in the *status register*.

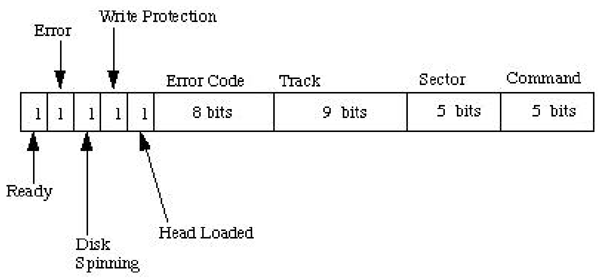


Figure 1. Disk control register

There are many of these registers and each register, or even each bit represents different meanings. Once engineers have the register values, they have to refer to the manual or supporting documents to understand what each bit field represents and the meaning of different values. It is tedious and easy to make mistakes. Sometimes, they develop special programs to parse the data. However, it takes time to implement the program and it is not that easy to use C to process arbitrary bits.

The web application developers face the same problem when writing network data processing code. Many network applications need to parse raw data according to standardized packet formats such as TCP/IP, FTP and SSL. Interpreting network data is the key to most important network applications including Web Server Testing, network traffic monitoring, network firewall checking and etc. Due to the complexity of networking protocol, and the bit-oriented feature of the networking data stream, it is complicated to write a program using C languages to interpret the data stream. Why C language is not sufficient in those areas is discussed in the following five aspects.

2.1 Byte-Alignment Constraints

The data type system of C language is byte-based, meaning it applies byte-alignment restriction. This assumes the storage media and the input/output routines have a minimum unit of one byte [4]. Programmers are conditioned to think of memory as a simple array of bytes [5].

A common C structure may appear as:

```
struct ABC {
    char      a;
    int       b;
    double    c;
} abc;
```

It specifies the physical layout of a three-field data stream: field “a” is one byte long, field “b” occupies four bytes (Type int usually is four bytes long, but depends on the system; it could be two bytes in MSDOS) and field ‘c’ owns an eight-byte sequential space. Types char, int and double are primitive types and as well as the other data types of C language, they are all integer multiples of one byte. All the operations are based on these “molecule” objects.

The real machine-level atomic unit is the bit. When the required granularity is less than a byte (1 byte = 8 bits) and we need bit-fields with arbitrary length, the byte-alignment can become a barrier for fast programming. If we want to use the C structure to specify an arbitrary length field, there is no ready-made data type that can give prominence to the domain. (The feature "typedef" only allows the user to introduce synonyms for types that could have been declared.)

Most networking data formats pack data as tightly as possible and can contain sufficient information using minimum space [3]. Take an IP packet for instance, the structure of IP header is shown in Figure 2.

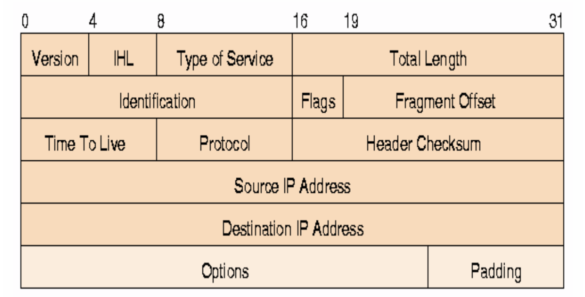
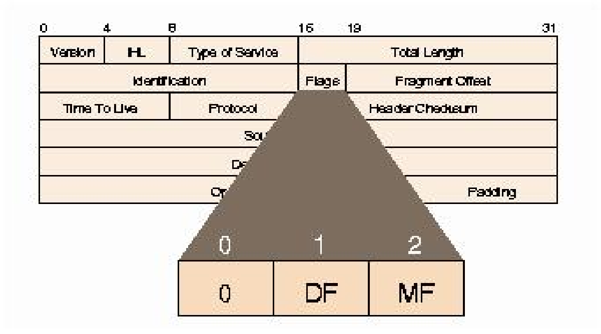


Figure 2. IP header structure

We can see some of the fields are not integer multiple of one byte: Version is 4 bits; IHL (Internet Header Length) is 4 bits; Flags (Various Control Flags) is 3 bits; FragmentOffset is 13 bits [6].

The content of Flags field is specified in Figure 3. Each bit has its individual meaning and the different bit value indicates certain functions. This three-bit field has a semantic meaning of "Various Control Flags", and each bit has a respective meaning as seen in the following list: Bit 0 is reserved and its value must be zero; Bit 1 is DF, which means "Do not Fragment" if that bit is set as one, otherwise it may Fragment. Similarly, bit 2 is MF, which means "More Fragment" if holding one bit.

We can use pointer arithmetic to access the memory and bit masking and shifting to achieve the bit operations. However, the pointer arithmetic and bit operations of C language are notorious for their tedious and error-prone nature. Moreover, the semantics end up buried in parsing code and the difficulty of reading the code can make the maintenance even harder [1].



Bit 0: reserved	**must be zero**
Bit 1: (DF)	**0 = May Fragment, 1 = Don't Fragment**
Bit 2: (MF)	**0 = Last Fragment, 1 = More Fragments**

Figure 3. Content of various control flags

2.2 Endianness Method Difference

The word "endianess" describes the method used to represent multi-byte integers in a computer system. There are two types of endianess: big endian refers to the method of storing the most significant byte of an integer at the highest byte address, and little endian is the opposite. Different endianess methods may apply for different machine's architecture, and may cause problems when application runs across different systems.

When referring to networking applications, network packet formats are independent of a machine's architecture but the network byte order and the host byte order can be different. When using C language to manipulate the packet, programmers must remember to convert any multi-byte numeric quantities between the two systems at all appropriate places [3]. For example, in the code net/ipv4 of Linux implement, the endianness related macros *ntoh** and *hton** together appear about 300 times.

C language also provides a practical feature called as bit fields which automatically packs the bit fields as compactly as possible and provides that the maximum length of the field is less than or equal to the integer word length of the computer [7]. However, bit fields lack of portability between platforms, because some bit field members are stored left to right others are stored right to left. That is also caused by endianness difference.

2.3 Field Dependency and Dynamic Typing

Some formats and especially some protocol headers can contain fields whose values or sizes depend on the value of a previous field. For instance, the Options field in the IP header can occupy between 0 to 40 bytes depending on the value of the previous field IHL (Internet Header Length) [3]. Since static types cannot represent variable-sized fields, such fields as Options are not supported by C *structs*. We need to use dynamic typing to solve this problem.

2.4 System-Dependent Word Size

Another small problem concerns the system dependent word size. As we mentioned in problem 2.1, the length of the Integer (type 'int') traditionally depends on the length of the system's Word type.

Thus in MSDOS it is 16 bits long whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes) [5]. The ambiguous type size may produce different results.

The question then is what about using existing libraries? Some similar libraries may provide functions that enable us to read a given file format. We have to learn the library interface first but it may lack the necessary features we need since the former developers did not foresee a need for different features. On the other hand, it may cost more in time and labour to convert similar programs according to different requirements. This decreases programming efficiency and can extend the development period.

So far, our problems are all related to the C language. Why are other high-level languages such as Standard ML, Perl not used? The reason is that it takes time to learn a new programming language. However, we may just need to use a fraction of its functions and a running environment.

Another frequently asked question is why not using XML. XML (eXtensible Markup Language) is a meta-language that is a way to define tag sets. To access an XML document file from a program, you can either parse the tag structure in your own code or using one of two standard APIs to invoke parsers to do it for you. The two APIs are DOM (Document Object Model) and SAX (Simple API for XML) [8]. These APIs are still based on byte type system and so far they did not focus on bit field's specification. So our future research direction is to embed XML concept into our project.

3 Method Prototype

The prototype of our project is shown in Figure 4. Originally, the user wants to pare the *Binary raw data* according to its *Specified Data Format* that is usually described in a human-language. And the user-expected result is the meaning and value of each bit field member; we call the output *Data Interpretation.* The *Specified Data Format* is on the left end of the figure and is associated with the binary raw data. The data format can be any user defined format or standard format (such as .bmp, or IP header format).

The box following the *Specified Data Format* is our core component, the *Data Format Scripting Language (DFSL)*. We can use this language to translate the documented data format into *Data Format Script*, which is the interface between the users and the computer. It is more straightforward for non-specialists to understand a scripting language.

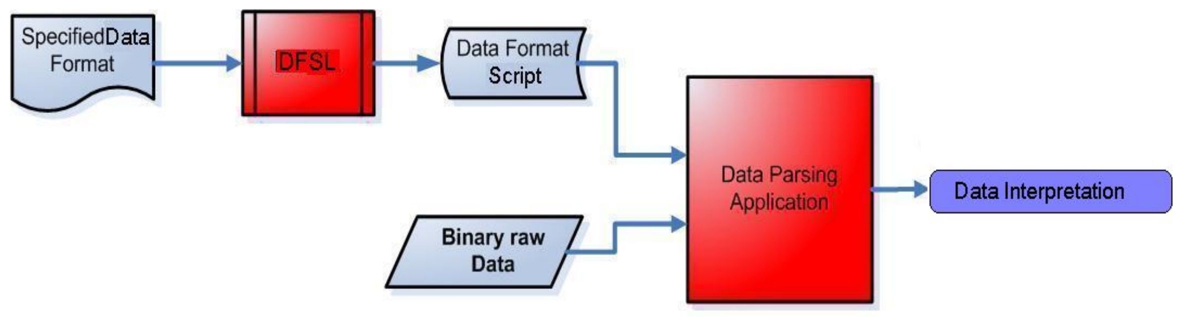


Figure 4. Data interpreting method prototype

Now we have the *Data Format Script* and *Binary raw data*. They are the two actual inputs for our generic *Data Parsing Application*, which is the main component of our project. This parsing application can parse the *Data Format Script* and execute any command in the script. The output is *Data Interpretation*, which includes the meaning of each bit field and its value.

This method is not limited to any particular data format. Once the data format is established, all the data with that format can be parsed.

4 Data format describing language

As a scripting language, DFSL is not a universal programming language, and has been developed for special usage (to interpret binary data). It is also not a substitute of C language. We only include minimum components to maintain its simplicity. It has special features that can simplify a user's specification, and some handy commands to facilitate bit field operations. We introduce these items in this section by examples.

4. 1 Layered Architecture Specification

When describing the layout of the data format, people usually use a tabular form such as we have seen in Section II, the IP header structure. In fact, the tabular form also represents a layered architecture (or called as hierarchical architecture). A representative example is the bitmap file (.bmp); its first layer is shown in Table 1:

Table 1. Bitmap file construct

Bitmap File Header
Bitmap Info Header
RGB Quad Array (or called as Optional Palette)
Colour-index Array (or called as Image Data)

Then each component has its individual structure that we call the sub-layer (see Table 2: the Bitmap file header is the first component in bitmap file construct):

Table 2. Bitmap file header construct

Bitmap File Header:		
Identifier	2bytes	File type, must be BM
File Size	1dword	Size of bitmap file
Reserved	1dword	Must be zero
Offset	1dword	Offset from beginning of file to the beginning of bitmap data

In our language, we use grouping and sequencing to specify the layered structure of the data format. Grouping can gather all the component members in the structure body, and the sequence of individual fields indicates the physical layout of the bit stream.

Structure definition has the form @label *:= { … }*: 1) the structure name is also called as a label which starts with a @; 2) the definition symbol is a colon with an equal sign; 3) the bound symbol are matched braces. The structure name is also used to contain real data, and its component fields can obtain bit fields from it. For instance, @IP_packet can get the data from a binary file with a file name "ip.dat":

```
   @IP_packet = getBinaryFile ("ip.dat")
   @IP_packet := {
    $version     = getBit 4;
    $IHL         = getBit 4 ;
    $TOS         = getByte ;
    $TLength     = getByte 2 ;
    $identif     = getByte 2 ;
    $reserved    = getBit ;
    $dontFrag    = getBit ;
    $moreFlag    = getBit ;
    $frageOffset = getBit 13;
    $ttl         = getByte;
    $protocol    = getByte;
    $cksum       = getByte 2;

    $src         = @ipAddress;
    $dest        = @ipAddress;
    $options     = @ipOptions;
    $padding     = byteString;
  }
  ...
```

Code 1. IP header structure

Take IP structure for example, we defines all fields of an Internet Protocol header structure (refer to Figure 1.) and groups those fields using braces under the label @IP_packet. The sequence of individual fields indicates the position of those fields in the real data. The code fragment is shown in Code 1.

Each field within the structure has a variable name that starts with the *$* symbol. We use an assignment statement which has the form *$name = <<right Value>>* to get the field value. The *right value* can be a real value or another *sub-layer* structure. As we can see from the sample code, most fields get the bits or byte value (e.g. *$name = getBit 4*); some of them are assigned a sub-layer structure name starting with @ (e.g. *$name = @structName*). Examples such as *$scr* (source IP address) and *$dest* (destination IP address) use the same structure *@ipAddress*. We then define the sub-structure outside of current structure as shown in Code 2.

```
  @ipAddress := {
   $first = getByte  ;
   $second = getByte ;
   $third = getByte ;
   $four = getByte ;
  }...
```

Code 2. Code for sub-layer structure

The hierarchical architecture means we can use grouping to specify the data structure and sequencing to describe the layout of the data. The sequence of terms determines the execution order. When meets a sub-layer label, the program will traverse the sub-layer structure its definition and resolves the fieldnames before returning and executes the next entry.

4.2 Read Bits

One of the advantages of using our language is that hides the tedious bit operations from the user. Because the data is regarded as a sequence of bit, we avoided endianness problem when read the bits. Its straightforward use achieves the desired bit fields by using command *getBit*, *getByte*, *seeBit and seeByte.* Those commands return integer value of the bit field.

The syntax of this series command is as follows:

- *getBit count*

Semantic: Read the number of "count" bits from current position (e.g. *getBit* 4); if offset is not given, default read one bit. The "count" can be a numeric number or an existing variable's value. We can see it as field dependency.

- *getBit @position , count*

Semantic: Read the number of "count" bits from the specified position (e.g. *getBit* @15, 3); if offset is not given, default read one bit. The "count" can be a numeric number or an existing variable's value.

- *getBit start ~ stop*

Semantic: Read bits from "start" position to "stop" position (e.g. getBit 15 ~ 9).

- *getByte count*

Semantic: Read the number of "count" bytes from the current position (e.g. getByte 3). The "count" can be a numeric number or an existing variable's value.

- *seeBit*

Semantic: *seeBit* shares the same syntax as *getBit*. It can only preview the bits without moving the bit pointer from the current position.

- *seeByte*

Semantic: *seeByte* shares the same syntax as *getByte*. It can only preview the bits without moving the bit pointer from the current position.

4.3 Constraint and Operations

As long as the bit stream is not exhausted, the data can be chopped and assigned to a field variable. However, there are always constraints on the data, and the extracted data field may also be operated.

We use a keyword *where* here to follow the current layer structure definition, and the statements braced in the *where* clause can do the real job, either be applying constraints on bit field or performing operations on the extracted data fields (see Code 3) while having no effect on the data layout. That is why we bring in *seeBit* function, which shares the same function as *getBit* but protects the continuity of the original data stream.

We can also define local variables for extra operations. As shown in Code 3, we defined a new variable *$classA* which received the value of the first bit of *@ipAddress* structure; if it is equal to zero, that means it is a Class A IP address and the following script will print its meaning and data values which are already extracted in the structure.

4.4 Casual Interpretation

We use the concept of layers to construct the layered architecture for complicated data format. However, for some straightforward formats, it will be more practical to interpret the result right after getting the bit field's value. So we allow the output routine be placed within the structure definition. For instance, PMD (Performance Motion Devices) has 16-bit status registers and each register has its own configuration (for a detailed description please refers to Appendix A).

As we can see in Table 3, the physical layout of this register is quite simple. There is no field dependency or complicated substructures. We can print out the bit field's semantic meaning and variable value right after we obtain its value.

```
@ipAddress := {
 $first = getByte  ;
 $second = getByte ;
 $third = getByte ;
 $four = getByte ;
} where {
  $classA = seeBit @31 , 1 ;
  if ( $classA == 0 )
  {
    print "Class A:" ;
    print "Network address: " , $first ;
    print "Host address:" + $second + "."
                  + $third + "." + $four ;
  }
  ...
}
```

Code 3. Fragment of "where" clause

Table 3. PMD configuration 3

PMD Configuration 3			
15 ~ 11	10 ~ 9	8	7
Tx Power Cutback Value	**Tx Power Cutback Mode**	**SBM Disable**	**Single Upstream Disable**
6	5	4	3~0
China loop	**OL Disable**	**ROL Disable**	**Hybrid Select**

Casual scripting is a practical feature in script writing. The user is free to add operational code once get the values they need. This is suitable for simple and straightforward format, and won't cause much trouble for future reading. The code fragment of interpreting the data in configuration 3 is shown in Code 4.

The programming style is quite similar to the C language but borrows ideas from other scripting language, such as Python or Perl to make it more convenient (e.g. implicit data type, simplified input/output routine). For instance, we use the "+" symbol to concatenate several components and the print command can print out any component in its scope. In the future we need to add more format specification to the output command.

4.5 Deformation of Switch Statement

A certain bits combination presents a certain meanings. The simplest situation is one bit, that like a switch of a certain function, when set to one means enable and set to zero means disable or vice versa. The more bits in a field, the more conditions it can represent.

Once the data is obtained, we know which condition is matched by checking the manual and our program can print out the desired interpretation. This function is always performed by a switch-case statement, and switch-case action is used in our script with a high frequency.

```
@PMD3 = 0x78ab ;
@PMD3 := {
$TxPowerValue = getBit 15 ~ 11 ;
#print "Bit 15 ~ 11: Tx Power
Cutback value (dB) = " + $TxPowerValue;

#print "Bit 10 ~ 9: Tx Power Cutback Mode" ;
$TxPowerMode = getBit 10 ~ 9 ;
   # 0 : "  0 : No Tx Power Cutback" ;
   # 1 : "  1 : Manual Tx Power Cutback" ;
   # 2 : "  2 : Automatic TA Power Cutback" ;
   # 3 : "  3 : Reserved" ;

$SBM = getBit @8 , 1 ;
......

$HybridSelect = getBit @3 , 4 ;

} where {
 $HybridMode = $HybridSelect -> seeBit @2 , 3 ;
    #  0 : " 0 : default setting" ;
    #  1 : " 1 : GPIO is in tri-state mode" ;
    #  default : "Reserved" ;
}
```

Code 4. Casual scripting

Because DFSL is a practical language for special usage, we deform the switch statement: 1) mark each condition with #; 2) compare the condition values with current field's value; we have a variable contains the current value. 3) Execute the desired actions when the condition is matched.

The sample code is shown below (Code 5.) *$HybridMod* is a three-bit field (technically it can have nine combinations). According to the configuration, 000 is the default setting, 001 means GPIO is in tri-state mode, while others are reserved.

```
$HybridMode = $HybridSelect -> seeBit @2 , 3 ;
    #  0 : " 0 : default setting" ;
    #  1 : " 1 : GPIO is in tri-state mode" ;
    #  default : "Reserved" ;
```

Code 5. Deformation of switch statement

Thus far we have briefly introduced the main feature of our language. As a scripting language still in development, the DFSL project is an open-ended project where we try to describe more data formats to the greatest extent. The associated application is generic software needed to parse this language. Using the application and the script, users can easily interpret a bit stream.

5 Related Work

Only a few related investigations have been carried out in this area. The most prominent are:

- **DataScript**. DataScript consists of two components: a constraint-based specification language to describe physical data layouts as types and a Java language binding that provides a simple programming interface to script binary data [3]. The specification language describes the data format by DataScript types: primitive types, set types, composite types, array types. Each field can associate with a constraint. Unlike our language, DataScripit cannot handle bit fields.

- **PacketTypes**. This specification language uses type system to match a network packet to a specified protocol. Types are used to eliminate the need of to write low-level code [2]. But it is only limited to some specified network protocol.

- **PADS** (Processing Arbitrary Data Streams). The goal of the PADS project is to simplify data stream analysis and to parse high-volume data streams such as web server logs [1]. However, it does not focus on bit field operations.

6 Conclusion

After investigating problems in a real industry environment, a practical solution is achieved to interpret raw data: regarding any kind of data as bit-stream, using a scripting language to describe the data format, and applying a software application to interpret the raw data according to the script.

This solution is relatively new and no similar application presently exists. The major contributions of this solution are:

- A simple, no compiler/link or software development system is needed. The only tool a user needs is a text editor.
- It is easy to use and requires no complicated programming.
- It is straightforward to read and very close to the data format or configuration table used to define the data format
- It saves development time.

As a result, this frees programmers from the tedious task of coding input/output routines, and eliminates the distractions from architecture-dependent problems making it possible to achieve faster development.

Reference

[1] K. Fisher and R. E. Gruber, “PADS: Processing arbitrary data stream”, *Proceeding of Workshop on Management and Processing of Data Streams*, San Diego, California, June 2003.

[2] G. Back, “DataScript – a specification and scripting language for binary data”, *Proceedings of the ACM Conference on Generative Programming and Component Engineering Proceedings (GPCE 2002)*, published as LNCS 2487. ACM, Pittsburgh, PA. October 2002. pp. 66-77

[3] P. J. McCann and S. Chandra, "Packet types: abstract specification of network protocol messages", *ACM SIGCOMM Computer Communication Review*, vol. 30, Issue 4, October 2000, pp. 321-333.

[4] J. Rentzsch, “Data alignment: Straighten up and fly right”, *IBM developerWorks*, March 2005, <http://www-128.ibm.com/developerworks/library/pa-dalign>

[5] J. Soulié, *The C++ Resources Network*, March 2005, <http://www.cplusplus.com/doc/tutorial/tut1-2.html>

[6] Suresh, “The IP Routing Protocol”, March 2005, <http://homepages.uel.ac.uk/u0219908/IPPacketStructure.htm>

[7] Dave Marshall, “Low level operators and bit fields”, March 2005, <http://www.cs.cf.ac.uk/Dave/C/node13.html>

[8] edikt, *Developer’s Guide*, ed 1.3, April 2005 <http://www.edikt.org/binx/docs/BinXDevGuide.pdf>

Appendix A

PMD Configuration 3															
Channel				**Reset = undefined**				**Read/Write**				**1A1CH**			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Tx Power Cutback Value					Tx Power Cutback Mode		SBM Disable	Single Upstream Disable	China loop	OL Disable	ROL Disable	Hybrid Select			

Name	**Bit**	**Description**			
Tx Power Cutback Value	15:11	**Tx Power Cutback Value** – The power cutback is a 5-bit binary value that ranges from 0 to 31 dB in 1 dB steps.			
Tx Power Cutback Mode	10:9	**Bit 10**	**Bit 9**	**Tx Power Cutback Mode**	
		0	0	No Tx Power Cutback – When this mode is selected, the total Tx power used is as specified in the standard and the annex used by the connection.	
		0	1	Manual Tx Power Cutback – When this mode is selected, the Tx power used is reduced by X dB from the standard specified value. The Tx power reduction factor, X, if implemented, is specified in the release note for a given DSP code version as either: A fixed value 2) The Tx Power Cutback field (if X is not specified in the release note for a given code)	
		1	0	Automatic TA Power Cutback – When this mode is selected and the actual downstream margin is greater than the maximum margin, the modem disconnects and cuts back the Tx power so as not to exceed the maximum margin. Once done, it reconnects.	
		1	1	Reserved.	
SBM Disable	8	Setting this bit to “1” disables the Single Bit Map mode in G.992.1 Annex Q.			
Single Upstream Disable	7	Setting this bit to ‘1’ disables single upstream.			
China loop	6	Setting this bit to ‘1’ improves the MOII china loop performance.			
OL Disable	5	Setting this bit to ‘1’ disables overlapping spectrum.			
ROL Disable	4	Setting this bit to ‘1’ disables reduced overlapping spectrum.			
Hybrid Select	3:0				
		Bit 2	**Bit 1**	**Bit 0**	**Hybrid Mode**
		0	0	0	Default setting, use GPIO to control line driver bias current.
		0	0	1	GPIO is in tri-state mode, line driver bias current is controlled by on board pull up/down circuit.
		x	x	x	Reserved.