Declarative Language FlexT for Analysis and Documenting of Binary Data Formats and Its Use for Data Reading Code Generation

Alexei Hmelnov

Matrosov Institute for System Dynamics and
Control Theory of Siberian Branch of Russian Academy of Sciences
Irkutsk, Russia
http://hmelnov.icc.ru/FlexT/

5 марта 2022 г.

Abstract

The language FlexT (Flexible Types) is intended for specification of binary data formats. The language is declarative and designed to be well understood for human readers. Its main elements are the data type declarations, which look very much like the usual type declarations of the imperative programming languages, but are more flexible. While the primary purpose of the language FlexT development was to make the binary data understandable by displaying them according to the format specifications, recently we have implemented the code generator, which can produce data reading code in some imperative languages from the specifications.

Requirements for Scientific Data Representation

- It becomes not enough to just obtain the data, process them and write some articles using the results of the processing
- It is also required to share the data with other researchers
- These researchers may be not only our contemporaries but also our descendants, living in a few decades from now
- The data, that we have stored for them, may become very precious, because they can't recollect the same data again (You cannot enter the same river twice)

The Binary File Formats vs the Text Ones

The binary data formats are much more space- and time-efficient, than the text-based ones. The main disadvantage of the binary data is that they look opaque for the users and it is hard to control their contents with a "naked eye". That's why programmers nevertheless often prefer to use the text formats, and among them the XML-based ones are of great popularity in spite of the fact that it becomes impossible for a human being to comprehend the extremely large text files.

Binary files

- space-efficiency
- simper and faster data reading/writing code
- random access (`Seek` operation), reading/writing of fragments

Text files

- transparency
- easy editing
- byte order agnostic (except for UTF-16 and the like)

The Choice of a New Generation is Text

The texts are usually of some markup kind like XML/JSON/YAML.

Reasons

- now developers often exchange efficiency of binary files for easier control over the correctness of text files contents;
- often text formats are based on the XML syntax, since there exist ready to use libraries and tools for this syntax that facilitate development of algorithms for reading/writing information;
- for XML it is possible to automate the control of the correctness of the file structure (using XML schemas).

My opinion

- It is better to store the information that needs to be edited frequently (for example, program settings) in small text files. Otherwise (when using binary files) we will have to write special editing forms for the binary files.
- It is better to avoid using text formats to exchange large amounts of information.
- To ensure the transparency of binary files and control their correctness we can use binary file format specifications.

The Goals of the Development of the Languages for Specification of Binary Data Formats

- Documenting of data formats
- Checking data for compliance to specification with error diagnostics
- Data reading code generation
- Meta-information for data processing/transfer libraries

Related works (1/2)

Tool	Purpose	Kind	Formats described	Bit types	Pointers	Variants	Human readable	Comment
BFF (Binary File Format Definition)	reverse engineering of the AutoCAD DWG	Grammatics	DWG	-	?	?	?	
DFDL (Data Format Description Language)	specifications of text and binary data used in GRID-systems, Open Grid, IBM	XML, simple records	tables in specialized exchange formats	-	-	?	-	
MFL (Message Format Language)	Specifications of exchange formats in WebLogic Integration, Oracle	XML, data streams	data in special exchange formats	-	-	+, supports optional fields	-	
NetPDL	specifications of network protocols and packet formats	XML, data streams	network packets	+, primitive	-	+	-	
BinPAC	specifications of network protocols, data reading code generation	modified Bison files, data streams	network packages	-	-	+	+	
EAST (Enhanced Ada SubseT)	developed by CCSDS to facilitate the exchange of information between space systems	Ada data types, data streams	sequentially transmitted space data	+	-	+, case by external value	+	simple expressions of single value
HUDDL	specification of hydrographic data, code generation	XML, data streams	hydrographic formats	?	-	?	-	

Related works (2/2)

Tool	Purpose	Kind	Formats described	Bit types	Pointers	Variants	Human readable	Comment
Advanced Language Processing Technology Applied to Digital Records	integration of heterogeneous information for military decision making, data reading code generation, USA military institute	attributed grammatics	sequential formats	?	-	?	+	
DataScript	generation of data reading libraries in Java	data type definitions	Java CLASS (no opcodes), DVI, ELF	+, primitive	-, has computable labels	+, unions with criteria	+	no updates since 2003
Miraplacid Binary DOM	implementation of universal binary and text data access library, DOM-like	data type definitions	21 format including text ones	-	+	-	+	Tree of data structures, selects current in hex dump
Kaitai Struct	binary data parser generation from specifications	∼JSON, sequential, has optional & repeating parts	36 formats	-	+	+	-	Now they have Switch type
Synalyze It!, Hexinator	Binary data viewer	XML, grammatics in Python	∼ 80 formats, mostly for macOS	+	+	+	-	Tree of data structures, colored dump parts
FlexT	Binary data parsing, data reading code generation	data type definitions	∼ 100 formats	+	+	+	+	

Related works: conclusions

- for many subject areas the need to use data format specifications is recognized
- sometimes the task is not to describe arbitrary formats, but only to facilitate processing of some their subset
- sometimes the task of describing arbitrary formats is set, but not all the constructions necessary to solve it are implemented
- often XML/JSON representation is chosen for specifications, which makes them hardly human-readable
- but in some projects, the requirement of the ease of perception of the text by human reader was clearly set
- a binary format specification is usually treated as a grammar, or as a sequential structure, or as a set of data types
- many specification languages have been created to facilitate development of data processing code
- developers of the tools rarely study related works
- the task of describing binary data is in demand

The Language FlexT

FlexT – **Flex**ible **T**ypes.
Flexible types – the types, that can adjust to the data (data type sizes and subitem offsets may vary).

The main goals of the language FlexT:

- provide the instrument, that can help us to explore and understand the contents of the binary files using format specifications (check and view data using specification);
- check whether the format specification is correct using the samples of the format data (check specification using data).

The FlexT data viewer makes the binary data transparent.

An Example of Binary Data – a DBF File

Hex dump of a small DBF file

```
LSKAT.DBF                                                    DOS      241     Col 0     100%
00000000:  03 60 0B 1C 05 00 00 00 │ 82 00 16 00 00 00 00 00
00000010:  00 00 00 00 00 00 00 00 │ 00 00 00 00 00 00 00 00
00000020:  4E 41 4D 45 00 69 B1 04 │ 94 1D 00 43 38 64 0A 00   NAME
00000030:  12 00 D4 6F 34 69 B2 21 │ F1 52 0A 00 D4 6F 44 69
00000040:  47 43 4F 4C 00 69 B1 04 │ 94 1D 00 4E 38 64 0A 00   GCOL
00000050:  02 00 D4 6F 34 69 B2 21 │ F1 52 0A 00 D4 6F 44 69
00000060:  54 4C 49 4E 45 00 B1 04 │ 94 1D 00 4E 38 64 0A 00   TLINE
00000070:  01 00 D4 6F 34 69 B2 21 │ F1 52 0A 00 D4 6F 44 69
00000080:  0D 00 20 87 A5 AC AF AE │ AB AE E2 AD AE 20 20 20     Землеполотно
00000090:  20 20 20 20 20 31 30 20 │ 20 88 91 91 8E 20 20 20          10   ИССО
000000A0:  20 20 20 20 20 20 20 20 │ 20 20 20 20 39 20 20 82                9   В
000000B0:  91 8F 20 20 20 20 20 20 │ 20 20 20 20 20 20 20 20   СП
000000C0:  20 31 35 30 20 8F E0 AE │ E7 A8 A5 20 20 20 20 20    150 Прочие
000000D0:  20 20 20 20 20 20 20 31 │ 34 20 20 90 A5 AC AE AD               14  Ремон
000000E0:  E2 20 20 20 20 20 20 20 │ 20 20 20 20 20 31 32 20   т                  12
000000F0:  1A
```

The information encoded in the file as shown by a specialized viewer

NAME	GCOL	TLINE
Землеполотно	10	
ИССО	9	
ВСП	15	0
Прочие	14	
Ремонт	12	

[▪]══ Структура D:...ODES\LSKAT.DBF ═

Имя	Тип	Длина	Десятки
NAME	Символ	18	
GCOL	Число	2	0
TLINE	Число	1	0

FlexT Specification of the DBF File Format

```
type
  TBinDate array[3] of Byte //date in
       binary format (YYMMDD)
  TDBF3FldKind enum Char (
    fkChar='C', fkNumeric='N', fkLog='L',
    fkDate='D', fkMemo='M'
  )
  TDBF3FldDsc struc
    array[11] of Char,0; Name //Name -
        ASCIIZ string
    TDBF3FldKind hType
    ulong DataP   //like Delphi Tag
    Byte Len      //field length in bytes
    Byte DecNum   //number of digits after
          dot
    Word MUsrRsrv1//Reserved for multiuser
          systems
    Byte WorkID   //ID of working area
    Word MUsrRsrv2//Reserved for multiuser
          systems
    Byte SetFldData//used by the command
        SET FIELDS
    array[8] of Byte Reserved
  ends
  PDataArray ^TDataArray near
  TDBF3Hdr struc
    Byte Ver //0x02-dBase II,0x03-dBase
        III
        //0x83-dBase III with Memo-
             fields
    TBinDate LastChangeDate
    ulong RecCnt  //Record count
    PDataArray HdrLen //Length of header
         in bytes
    Word RecLen        //Length of record
         in bytes
    (array[20]of Byte) Reserved
  ends

  TDBF3HdrWithFields struc
    TDBF3Hdr H
    array[(@.H.HdrLen-@.H:Size-1) div
      32] of TDBF3FldDsc Fields
  ends
data
  0x0000 TDBF3HdrWithFields Hdr
type
  TFieldData array[Hdr.Fields[#].Len]of
                Char
  TFieldsData array of TFieldData :
              [@:Size=Hdr.H.RecLen-1]
  TRecData struc
    Char F
    TFieldsData D
  ends
  TDataArray array[Hdr.H.RecCnt] of
      TRecData
```

The Parse Results for the DBF File

```
00:Hdr: TDBF3HdrWithFields = (
  H:(Ver:dBase_III{03}; LastChangeDate:(Y:96; M:11; D:28); RecCnt:00000005; HdrLen:0082;
    RecLen:0016; Reserved:
   (0:00,1:00,2:00,3:00,4:00,5:00,6:00,7:00,8:00,9:00,10:00,11:00,12:00,13:00,14:00,
    15:00,16:00,17:00,18:00,19:00));
  Fields: (
    0:(Name:'NAME'; hType:fkChar{'C'}; DataP:000A6438; Len:12; DecNum:00;
       MUsrRsrv1:6FD4; WorkID:34; MUsrRsrv2:B269; SetFldData:21;
       Reserved:(0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44); InMDX:69),
    1:(Name:'GCOL'; hType:fkNumeric{'N'}; DataP:000A6438; Len:02; DecNum:00;
       MUsrRsrv1:6FD4; WorkID:34; MUsrRsrv2:B269; SetFldData:21;
       Reserved:(0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44); InMDX:69),
    2:(Name:'TLINE'; hType:fkNumeric{'N'}; DataP:000A6438; Len:01; DecNum:00;
       MUsrRsrv1:6FD4; WorkID:34; MUsrRsrv2:B269; SetFldData:21;
       Reserved:(0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44); InMDX:69),
    3:0D))
    81:00 ¦.¦
82:Hdr.H.HdrLen^: TDataArray =  (
  0:(F:' '; D:  (0:'Земполотно         ',1:'10',2:' ')),
  1:(F:' '; D:  (0:'ИССО               ',1:' 9',2:' ')),
  2:(F:' '; D:  (0:'ВСП                ',1:'15',2:'0')),
  3:(F:' '; D:  (0:'Прочие             ',1:'14',2:' ')),
  4:(F:' '; D:  (0:'Ремонт             ',1:'12',2:' ')))
    F0:1A ¦.¦
```

The Advantages of Specifications

in comparison with the possible sources of information about a file format:

documentation The vast majority of the format specifications written in natural language contain errors and ambiguities, which can be detected and fixed by trying to apply the various versions of specification to the real data to find the correct variant of understanding of the format description;
Inaccuracy, **incompleteness**, **ambiguity**

source code The information about a file format may also be obtained from the source code of a program that works with it. But the code contains a lot of unessential details of some concrete way of data processing. So, the resulting specification will be much more concise and understandable;
The information about format is **intermixed** with file I/O and data processing operations

data samples We have a successful experience of reverse engineering of some file formats using just the samples of data.
Initially specification is **missing**

Features of the FlexT Language

- The major part of the information about a file format is represented by the data type declarations
- In contrast to the data types of imperative programming languages, the FlexT data types can contain data elements, the size of which is determined by the specific data represented in the format. Thus, we can say that the types flexibly adjust to the data
- After defining the data types, it is required to specify the placement in memory of some data elements which have some of these types (*declare variables*)
- The language syntax was chosen to be well-understandable by human reader

Dynamic Data Types vs Static Ones

<table>
<tr><th>Type</th><th>Static</th><th>Dynamic</th></tr>
<tr><td>Size</td><td>fixed</td><td>data-dependent</td></tr>
<tr><td>Sub-item offsets</td><td>fixed</td><td>data-dependent</td></tr>
<tr><td>Usage</td><td>types of variables</td><td>types of constants</td></tr>
<tr><td>Pascal examples</td><td><pre>TName=array[0..31]
    of Char;
TId = string[32];</pre></td><td><pre>const CP: PChar =
    'Hello';
var S: String;
  ...
S := 'Hello';</pre></td></tr>
</table>

RTTI: examples of dynamic data types

RTTI – RunTime Type Information

Data type declaration in Pascal (in `TypInfo` unit)	The function for fetching the `TypeData` field	`FlexT` specification of the type

`PTypeInfo = ^TTypeInfo;`
`TTypeInfo = record`
`  Kind: TTypeKind;`
`  Name: ShortString;`
` {TypeData: TTypeData}`
`end;`
`PTypeData=^TTypeData;`
`TTypeData=packed record`
`  case TTypeKind of`
	`function GetTypeData(TypeInfo:`
	`  PTypeInfo): PTypeData;`
	`asm`
	`  XOR EDX,EDX`
	`  MOV DL,[EAX].TTypeInfo.`
	`      Name.Byte[0]`
	`  LEA EAX,[EAX].TTypeInfo.`
	`      Name[EDX+1]`
	`end;`
		`TTypeInfo struc`
		`  TTypeKind Kind`
		`  Str Name`
		`  TTypeData TypeData`
		`ends`

Parameters and properties of data types

- Data types can have a number of properties (depends on the kind of the type).
- For example, the size and the number of elements are the properties of arrays, and the selected case number is the property of variants.
- Each data type has the property `Size`
- The values of the properties can be specified in the statements of type declaration, and also by expressions that compute the value of this property using the values and properties of the nested data elements and/or the values of the parameters of the type in the block of statements.
- The parameters in the type declaration represent the information that needs to be specified additionally when the type is used (*called*).
- Almost all the FlexT data types have their bit-oriented versions.

Specifics of the FlexT language (1/2)

1. Sub-elements of variable size

```
FldNameTbl: array[@.numFields] of pchar;
```

2. Parameters and properties of data types

```
THdrData(Kind,Cnt) case THdrValType(@:Kind) of
  CHAR: array[@:Cnt] of Char
  ............................
endc
```

3. Qualifiers of properties, parameters and sub-elements in expressions
   - `V[i]` – array element;
   - `V.A` – record field;
   - `V.0x15`, `V.'asoc'` – case of variant;
   - `V:Size` – parameter or property;
   - `@` – the instance of the type being defined (let's call it *Self* or *this*);
   - `V@` – the parent of `V` (inside which it is defined), available for nested types only, data type of `V@` is known;
   - `V:@` – *owner* of `V` (the variable that immediately contains `V`), data type of `V:@` is unknown;
   - `V:#` – ordinal number of `V` inside its owner;
   - `&V` – *address≡file offset* of `V`, integer value.

Specifics of the FlexT language (2/2)

4. Blocks of additional data type information:
   - Block of statements `:[@.offset:Cnt=@.count]`
   - Block of assertions (correctness conditions) `:assert[@.Op>=0x80]`
   - Display block `:displ=(INT(2*@))`
   - Auto-naming block `:autoname=(@.tag)`
   - Definition of additional computable property `:let Val=(@.0)exc(@.1)`
5. Type calls `PSubSecData(Kind=@.subsec)lfo`

FlexT data types (1)

Type	Example	Description/purpose
Integer[a]	`num-(6)`	differ by the size and the presence of a sign
Empty[a]	void	the type of size 0, marks a place in memory
Characters[a]	char, wchar, wcharr	In the selected character encoding or Unicode with the byte orders LSB or MSB
Enumeration[a]	`enum byte (A=1,B,C)`	specifies the names of constants of the basic data type

Term enumeration	`enum TBit8 fields(`
	`  R0: TReg @0.3,...)`
	` of(`
	`  rts(R0) = 000020_,...)`
		simplifies description of encoding of machine instructions, specifies the bit fields, the presence of which is determined by the remaining bits of the number

Set of bits[a]	`set 8 of (` `  OLD ^ 0x02, ...)`	gives the name to bits, the bits can be designated by their numbers (the symbol '=' after the name) or masks (the symbol '^')

Record[a]	`struc`
	`  Byte Len`
	`  array[@.Len]of Char S`
	`ends`
		Sequential placement in memory of named data elements, which may have different types

Variant[a]	`case @.Kind of`
	`  vkByte: Byte`
	`else ulong`
	`endc`
		Selects the content type by the external information

FlexT data types (2)

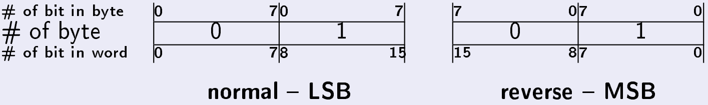
Type	Example	Description/purpose
Type check[a]	`try FN: TFntNum Op: TDVIOp endt`	Selects the content type by internal information (the first type, which satisfies its correctness condition)
Array[a]	`array[@.Len]of str` `array of str ?@[0]= 0!byte;`	Consecutive placement of the constituent parts of the same type in memory (the sizes of which may vary). It may be limited by the number of elements, the total size, or the stop condition
Raw data[a]	`raw[@.S]`	Uninterpreted data, which is displayed as a hex dump
Alignment[a]	`align 16 at &@;`	Skips unused data to align at the relative to the base address offset, which is a multiple of the specified value
Pointer	`^TTable near=DWORD, ref=@:Base+@;`	Uses the value of the base type for specifying the address (for files – the file offset) of the data of the referenced type in memory
Forward declaration[a]	`forward`	allows to describe cyclic dependencies between data types
Machine instructions	`codes of TOpPDP ?(@.Op >=TWOpCode.br)and...;`	machine code disassembling

[a]Supported by the reader code generator

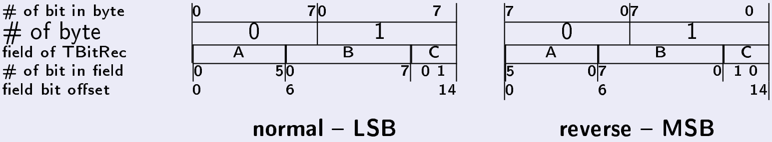
A bit record

```
type bit
TBitRec struc
  num+(6) A
  num+(8) B
  num+(2) C
ends
```

The order of bytes and bits in a 2-byte word



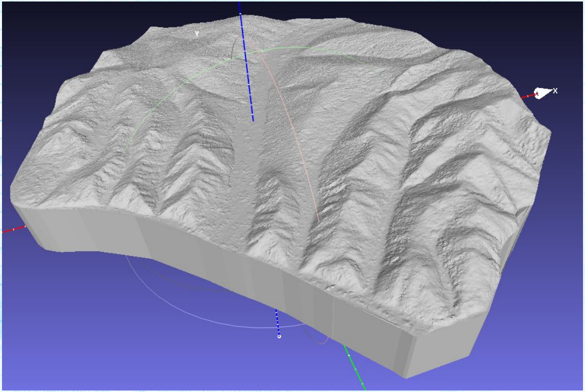
Bitwise placement of the TBitRec fields



STL files

STL - STereo-Lithography:

- The main format for 3D models
- Extremely simple



The STL format specification in FlexT and parse results

```
data
0 array[5] of char Hdr0
assert not(Hdr0='solid');

include Float.rfi

type
TSTLPoint array[3] of TSingle
TSTLFace struc
  TSTLPoint Normal
  array[3] of TSTLPoint Vertex
  Word Attr
ends

data
5 array[75] of char Hdr1
80 ulong Count

assert 84+Count*TSTLFace:Size=FileSize;

data
84 array[Count] of TSTLFace Faces
```

Parse results

```
0000000:Hdr0: #Tfl_12_c3 = 'STL f'
0000005:Hdr1: #Tfl_119_c3 = 'ile generated by TIN
Smith                                                    '
0000050:Count: ulong = 000D445C
0000054:Faces: #Tfl_126_c4 =  (
  0:(Normal: (0:0.125387370586395,1:-0.228255271911621,
      2:0.965493440628052);
    Vertex: (
      0: (0:-14.1680641174316,1:47.9845542907715,
        2:10.9498138427734),
      1: (0:-14.1677465438843,1:48.1474494934082,
        2:10.9882831573486),
      2: (0:-14.3192071914673,1:47.9848518371582,
        2:10.9695129394531)); Attr:0000),
  1:(Normal: (0:0,1:0,2:-1);
    Vertex: (0: (0:-14.1680641174316,1:47.9845542907715,2:0),1:
       (0:-14.3192071914673,1:47.9848518371582,2:0),
      2: (0:-14.1677465438843,1:48.1474494934082,2:0)); Attr:0000),
  2:(Normal: (0:-0.0630344226956367,1:-0.224894672632217,
      2:0.972342014312744);
    Vertex: (
      0: (0:-14.0169200897217,1:47.984260559082,
        2:10.9595441818237),
      1: (0:-14.0166034698486,1:48.147155761718 8,
        2:10.9972410202026),
      2: (0:-14.1680641174316,1:47.9845542907715,
        2:10.9498138427734)); Attr:0000),
  3:(Normal: (0:0,1:0,2:-1);
    Vertex: (0: (0:-14.0169200897217,1:47.984260559082,2:0),
      1: (0:-14.1680641174316,1:47.9845542907715,2:0),
      2: (0:-14.0166034698486,1:48.1471557617188,2:0)); Attr:0000),
  4:(Normal: (0:0.0398440323770046,1:-0.30687946081l615,
```

The HiSCORE custom file format specification

```
type bit
TBit5 num+(5):displ=(int(@))
TBit6 num+(6):displ=(int(@))
TDNS num+(7):displ=(int(@*10))
TBit10 num+(10):displ=(int(@))
TTime struc
  TDNS dns
  TBit10 mks
  TBit10 mls
  TBit6 s
  TBit6 m
  TBit5 h
  num+(20) rest
ends:displ=(int(@.h),':',int(@.m),'''',int(@
      .s),'.',int(@.mls),'␣',int(@.mks),'␣',
      int(@.dns))

type
TVal num+(2):displ=(int(@))
TTrackInfo struc
  word offset //track offset
  TVal N //длина N
  array[@.N]of TVal Data //N байт - track
       data
ends:displ=('[',ADDR(&@),']',@)

TPkgData(Sz) struc
  array[9]of TTrackInfo Tracks
  ulong Stop //4 байта - FF FF FF FF -
       package end
  raw[] rest //Just in case
ends:[@:Size=@:Sz]:assert[@.Stop=0xFFFFFFFF]
```

```
TPkgHdr struc //Package header (24 байта):
  word idf //data type id = 3008
  word NumBytes //package size (without the
       24 bytes of the header)
  ulong NumEvent //event counter number
  ulong StopTrigger //position of stop
       trigger in DRS counts
  TTime EventT //event time
  word IP //IP adress
  word NumSt//Number of station
  TPkgData(@.NumBytes) Data
ends:assert[@.idf=3008]

data
0 array of TPkgHdr:[@:Size=FileSize] Hdr
```

```
000000:Hdr: #Tf2_147_c3 =  (
  0:(idf:0BC0; NumBytes:2128; NumEvent:00002F32;
    StopTrigger:0000014B; EventT:13:19'5.436 166 19; IP:0000;
    NumSt:0000;
    Data:(Tracks: (
        0:[18](offset:00A0; N:400;
          Data: (0:6693,1:6976,2:6914,3:6960,4:6717,5:6805,6:7074,
            7:7178,8:7071,9:7278,10:6986,11:7237,12:6934,13:7093,
            14:6821,15:6945,16:6810,17:7041,18:6909,19:6998,20:7079,
            21:7054,22:7003,23:6983,24:7067,25:7050,26:6951,27:7120,
            28:6899,29:6878,30:6891,31:7069,32:6909,33:7040,34:6900,
            35:6879,36:6817,37:6897,38:6926,39:6965,40:7022,41:6878,
            42:7059,43:6885,44:7122,45:6935,46:7022,47:6920,48:7142
```

read_hisc.c - 278 lines, Hiscore.rfi - 47 lines

Weather data in the MM5 format

One of the possible sources of information about a file format is the source code, which can process it.

- The advantages of the source code over the descriptions in natural language are its proved correctness (the code can indeed process the data) and the lack of ambiguity.
- So, it may seem that understanding a file format by analyzing a source code for its processing will always be easy and preferable to reading the specifications in natural language.

BUT in our experience of FlexT usage we have an indicative example, which demonstrates, that sometimes it may be very hard to understand the file format using the source code.

The file format MM5 was used for representation of the weather forecast data, computed by some Earth climate models. The data contain multidimensional grids for the various climate values (temperature, pressure, wind speed and so on). It was required to read the MD5 file to do something useful with it (say, compute the isolines).

Excerpts from the file readv3.f for reading the MM5

```
program readv3  ! This utility program is written in free-format
    Fortran 90.
...
  integer, dimension(50,20) :: bhi
  real, dimension(20,20) :: bhr
  character(len=80), dimension(50,20) :: bhic
  character(len=80), dimension(20,20) :: bhrc
  character(len=120) :: flnm
  integer :: iunit = 10
...
  print*, 'flnm = ', trim(flnm)
  open(iunit, file=flnm, form='unformatted', status='old', action='
      read')
...
  read(iunit, iostat=ierr) flag
  do while (ierr == 0)
     if (flag == 0) then
        read(iunit,iostat=ier) bhi, bhr, bhic, bhrc
...
        call printout_big_header(bhi, bhr, bhic, bhrc)
     elseif (flag == 1) then
...
```

Excerpts from the MM5 data version 3 format specification in FlexT and parse results

```
TBHi array[50] of array[20] of i4
Tbhr array[20] of array[20] of
    TReal

TComment array[80] of Char,<0x20;

TBHiC array[50] of array[20] of
    TComment
TbhrC array[20] of array[20] of
    TComment

TBigHeader struc
  u4 BHSize //Size of Data -
      added by Fortran write
  TBHi BHi
  Tbhr bhr
  TBHiC BHiC
  TbhrC bhrC
  u4 BHSize_ //Size of Data -
      added by Fortran write
ends:assert[@.BHSize=@:size-8,@.
    BHSize_=@:size-8]
```

```
D:(BHSize:0001CB60;
  BHi: (0: (0:11,1:1,2:6,3:0,4:52,5:52,
         6:1,7:0,8:52,9:52,10:0,...),
    1: (0:3,1:2,2:16,3:2,4:-999,5:-999,
         6:-999,7:-999,8:-999,9:-999,...),
    ...
    49: (0:-999,1:-999,2:-999,3:-999,
         4:-999,5:-999,6:-999,7:-999,...);
  bhr: (0: (0:9000,1:56.5,2:85,
         3:0.71556681394577,4:60,...),
    1: (0:21600,1:10000,2:-999,3:-999,
         4:-999,5:-999,6:-999,7:-999,...),
    ...
    19: (0:-999,1:-999,2:-999,...);
  BHiC: (0: (0:OUTPUT FROM PROGRAM MM5 V3,
      1:TERRAIN VERSION 3 MM5 SYSTEM FORMAT EDITION NU
      2:TERRAIN PROGRAM VERSION NUMBER,
      3:TERRAIN PROGRAM MINOR REVISION NUMBER,...));
  bhrC: (0: (0:COARSE DOMAIN GRID DISTANCE (m),
      1:COARSE DOMAIN CENTER LATITUDE (degree),...));
  BHSize_:0001CB60))
```

Advantages of formal specifications

- compactness and absence of information that is not related to the methods of storing data, which facilitates their perception by a human reader;
- verifiability by their usage for parsing valid data;
- they may be used for localizing errors in generated files.

Verification of data conformity to specification

The result of parsing data using specification can reflect the value of each bit of the source file. Parsing data according to specification can be used to:

- validating data against the specification (like it is done for XML using XML Schemas);
- verify the specification for their compliance to valid format data;

A specification to be tested may allow data that only partially conforms to the full specification.

Refinement of specification in FlexT during its development

1. Describe in `FlexT` a fragment of the format as we understand it now;
2. Apply the current version for parsing a valid format file;
3. If the parse result has errors fix the specification, go to 2;
4. If it is not finished, got to 1

data Format Reverse Engineering

1. is the ultimate case of specification refinement;
2. should be applied when we have enough samples of data;
3. the best case: we have data generator, e.g. a compiler

Generation of data reading code

- The format specifications are required to write a correct program, that should work with the files of the format;
- Because the FlexT language data types look similar to that of imperative languages, it is possible to immediately use some parts of specification to declare the data types, constants, and so on, which are required to write the data processing code. Anyway the process of writing the code manually is still time-consuming and error-prone;
- So, we have implemented the code generator, which can automatically produce the data reading code in imperative languages from the FlexT specifications;
- By its expressive power the FlexT language outperforms the other projects developing the binary format specifications, so the task of code generation for the FlexT specifications is rather nontrivial;
- By now we have implemented the code generation for the most widely used FlexT data types, but some complex types are not supported yet.

Example of translation of FlexT expression

FlexT specification of polygon/polyline data in Shape file format

```
TArcData struc
  TBBox BBox
  long NumParts
  long NumPoints
  array[@.NumParts] of long Parts
  array[@.NumParts] of struc
     TXPointTbl((@@@.Parts[@:#+1] exc @@@.NumPoints)-@@@.Parts[@:#]) T
   ends Points
ends
```

Generated Pascal code, which provides accessor for the field T

```
function TTArcData_Sub1Accessor.T: TTXPointTblAccessor;
var
  i0: Integer;
  ndx0: Integer;
begin
  if not Assigned(FT) then begin
    ndx0 := Index+1;
    if (ndx0>=0)and(ndx0<TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.
         Count) then
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.Fetch(ndx0)
    else
      i0 := TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).NumPoints;
    FT := TTXPointTblAccessor.Create(Self,0,0,i0-
      TTArcDataAccessor(TTArcData_Sub2Accessor(Parent).Parent).Parts.Fetch(Index));
  end;
  Result := FT;
end ;
```

Generation of the test application

- The first thing any programmer will want to do after generation of a data reader is to test whether it works well.
- To perform the test it is required to write some application, which will use the data reader somehow.
- The most obvious and illustrative task here is to print using the data reader.
- After creating manually several test programs of this kind we have found that the process is rather tedious and that it should be automated.
- So, we have developed the algorithm, which automatically generates the test code.
- The test program generated together with the data reader allows to immediately check the reader.
- Of no less importance is the fact that the source code of the program demonstrates the main patterns of data access using the reader.

Fragments of the test application code in C++, immediate write style

```
std::unique_ptr<TSHPReader> must_free_Reader(new TSHPReader(FN));
Reader = must_free_Reader.get();
if (!AssertTShpHeader(Reader->Hdr(),Reader))
  exit(2);
cout<<"Hdr:"<<endl;
cout<<sIndent<<"Magic: "<<Reader->Hdr()->Magic.Value()<<endl;
...
cout<<sIndent<<"FileLength: "<<Reader->Hdr()->FileLength.Value()<<endl;
cout<<sIndent<<"Ver: "<<Reader->Hdr()->Ver<<endl;
...
cout<<"Tbl:"<<endl;
for (i=0; i<Reader->Tbl()->Count(); i++) {
  V = Reader->Tbl()->Fetch(i);
  cout<<sIndent<<"["<<i<<"]:"<<endl;
  cout<<sIndent<<"RecNo: "<<V->RecNo()<<endl;
  cout<<sIndent<<"Len: "<<V->Len()<<endl;
  if (!V->Data()->GetAssert())
    exit(2);
  cout<<sIndent<<"Data:"<<endl;
  cout<<sIndent<<"ST: "<<TShapeTypeToStr(V->Data()->ST())<<endl;
  cout<<sIndent<<"SD:"<<endl;
  switch ( (TShapeRecData_Sub0_Case)V->Data()->SD()->hCase() ) {
    case hcPoint:
      cout<<sIndent<<"Point:"<<endl;
      cout<<sIndent<<"X: "<<V->Data()->SD()->cPoint()->X<<endl;
      cout<<sIndent<<"Y: "<<V->Data()->SD()->cPoint()->Y<<endl;
      break;
    ...
    case hcMultiPointZ:
      cout<<sIndent<<"MultiPointZ:"<<endl;
      ...
      cout<<sIndent<<"Points:"<<endl;
      for (i13=0; i13<V->Data()->SD()->cMultiPointZ()->A()->Points()->Count(); i13++) {
        V13 = V->Data()->SD()->cMultiPointZ()->A()->Points()->Fetch(i13);
```

Fragments of the test application code in Pascal, procedural style

```
procedure printTClassFile_Sub0(const sIndent: String; AV: TTClassFile_Sub0Accessor);
var
  i: Integer;
  V: TCp_infoAccessor;
begin
  for i:=0 to AV.Count-1 do begin
    V := AV.Fetch(i);
    Writeln(sIndent,'[',i,']:');
    printcp_info(sIndent+'␣␣',V);
  end;
end ;
...
procedure printTClassFile(const sIndent: String; AV: TTClassFileAccessor);
var
  sIndent1: String;
begin
  Writeln(sIndent,'minor_version:␣',AV.minor_version);
  Writeln(sIndent,'major_version:␣',AV.major_version);
  Writeln(sIndent,'C_pool_count:␣',AV.C_pool_count);
  Writeln(sIndent,'C_pool:');
  sIndent1 := sIndent+'␣␣';
  printTClassFile_Sub0(sIndent1,AV.C_pool);
  ...
end ;
...
    Reader := TClaReader.Create(FN);
    try
      Writeln('magic:␣',Reader.magic);
      Writeln('Hdr:');
      printTClassFile('␣␣',Reader.Hdr);
    finally
      Reader.Free;
    end;
```

Conclusion

- We have considered the possible options, which should be examined when selecting a file format for scientific data representation.
- The formal specifications of binary file formats, especially for the custom ones, are very important, because the natural language specifications are ambiguous, and it may be hard to fetch the data format information from the source code.
- The language FlexT allows to write compact, human-readable and powerful specifications, which allow to check the correctness of data and resolve the ambiguities in the understanding of the other kinds of information about file formats.
- It is also possible to generate from the FlexT specification the data reading code and the code of the application, that can immediately test the generated reader by printing the whole content of a binary file according to the specification using the reader.
- The current level of capabilities of the code generator is well characterized by that it have successfully produced a full-featured data reader code for the well-known for the GIS community Shape file format. The FlexT specification of the Shape format takes approximately 180 lines of code. The code generator have produced 1570 lines of the reader code, and 375 lines of the test program.
- The algorithm developed was also used for generation of the data readers for some custom scientific file formats.