On the use of specifications of binary file formats for analysis and processing of binary scientific data

Alexei Hmelnov

Matrosov Institute for System Dynamics and
Control Theory of Siberian Branch of Russian Academy of Sciences
Irkutsk, Russia
`http://idstu.irk.ru`

3 апреля 2019 г.

Abstract (1)

The data collected during various kinds of scientific research may be represented both by well known binary file formats and by custom formats specially developed for some unique device. While thorough understanding of the file format may be required for the former case of the well known format, for the latter case of custom formats it is of critical importance. For the custom formats usually only few people know how they are organized, and this expertise can easily be lost (man is mortal and death is sudden).

Abstract (2)

The language FlexT has been developed for specification of binary file formats. Its primary purpose is to be able to view the binary data and to check that the data conform to the specification, and that the specification conforms to the data samples available. As a result of the tests we can be sure, that the specification is correct. The FlexT specifications doesn't contain any surplus information besides from that about the file format. They are compact and human readable. We have also developed the algorithm for data reading code generation from the specification.
In the report we'll consider some FlexT language details and the experience of its application to specification of some scientific data formats.

The outcomes of scientific research

- Articles:
  - summarize the main results of the analysis of the collected data
  - the results depend on the methods chosen, their parameters, and some other subjective factors
- Scientific data:
  - Other researchers are interested in the access to the obtained data (both raw original and processed) to be able to
    - ★ check the research results
    - ★ try their own way of the data analysis
    - ★ compare the data to their own ones
    - ★ use the data as the input for the other research projects
  - The concept of scientific data life-cycle allows to support this demand

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)

Possible options for the scientific data format choice decisions

Level of customization:

- well-known file formats with ready to use data processing libraries and software (JPEG,TIFF,DBF,CSV,SHP,...)
- data exchange frameworks with the data format setup/data processing/code generation libraries, software
- “Universal“ formats with the data format setup/data processing/code generation libraries, software
- custom file formats

Usage:

- internal
- exchange

File type:

- text-based, XML
- binary

Data exchange frameworks

Allow to develop custom formats, which should fit the Procrustes' bed of the framework limitations.
Some examples:

- DFDL (Data Format Description Language) - text and binary data in GRID systems, prevalently tables, specifications in XML, product: IBM Integration Bus
- MFL (Message Format Language) - sequential binary data with repetitions and optional fields, specifications in XML, product: WebLogic Integration by Oracle
- EAST (Enhanced Ada SubseT) developed by Consultative Committee for Space Data Systems (CCSDS), sequential binary data

“Universal“ formats

Allow to store wide (but still limited) range of information.
Some examples:

- XML - text based, may use XML schemata and libraries/code generation software to simplify data processing.
- Hierarchical Data Format (HDF) is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. Originally developed at the National Center for Supercomputing Applications, now supported by The HDF Group. Libraries for C,C++,Java, MATLAB, Scilab, Octave, Mathematica, IDL, Python, R, Fortran, and Julia. Main goal: portable scientific data format. HDF is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF4 file can hold a mix of related objects which can be accessed as a group or as individual objects. Users can create their own grouping structures called "vgroups."HDF5 simplifies the file structure to include only two major types of object:
    - Datasets, which are multidimensional arrays of a homogeneous type
    - Groups, which are container structures which can hold datasets and other groups
- Protocol Buffers (by Google) are a method of serializing structured data. Widely used by neural network libraries. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for writing or reading a stream of bytes that represents the structured data. The resulting files are not self-contained and require the structure specification (or, better, the code generated from it) to read the files of a particular kind.

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.

The language FlexT

FlexT – **Flex**ible **T**ypes.
Flexible types – the types, that can adjust to the data (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.

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;

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;

data samples
: We have a successful experience of reverse engineering of some file formats using just the samples of data.

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
- The language syntax was chosen to be well-understandable by human reader

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 using the values of the parameters of the type.
- 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 the bit-oriented versions.

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

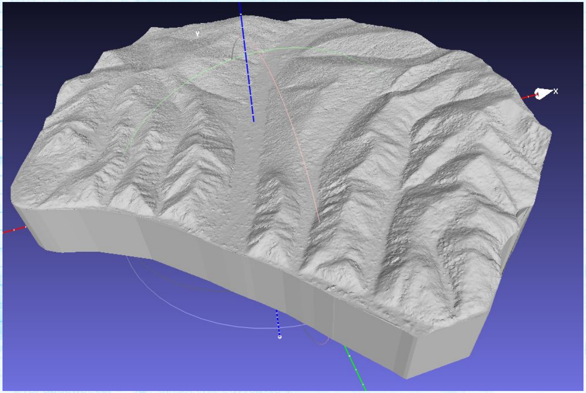
Machine instructions	`codes of TOpPDP ?(@.Op`
	`  >=TWOpCode.br)and...;`
		machine code disassembling

[a]Supported by the reader code generator

STL files

STL - STereo-Lithography:

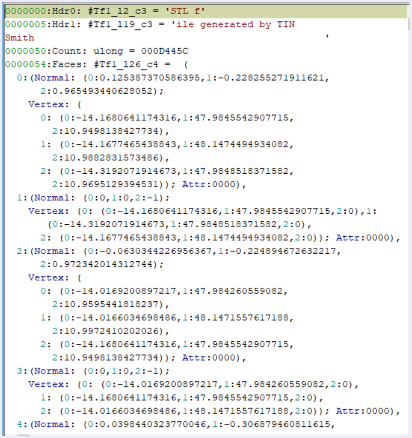
- The main format for 3D models
- Extremely simple



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

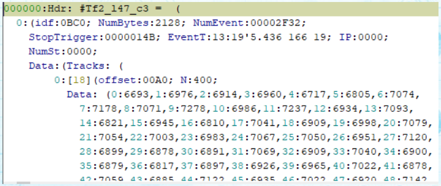


```
0000000:Hdr0: #Tfl_12_c3 = 'STL f'
0000005:Hdr1: #Tfl_119_c3 = 'ile generated by TIN
Smith                                                   '
0000050:Count: ulong = 000D445C
0000054:Faces: #Tfl_126_c4 =  (
  0:(Normal: (0:0.125387370586395,1:-0.228255271911621,
      2:0.965493440628052);
    Vertex: (
      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:-14.0166034698486,1:48.1471557617188,
        2:10.997241020202 6),
      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:-0.30687946081161 5,
```

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

```fortran
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 FORM
      2:TERRAIN PROGRAM VERSION NUMBER,
      3:TERRAIN PROGRAM MINOR REVISION NU
  bhrC: (0: (0:COARSE DOMAIN GRID DISTANC
      1:COARSE DOMAIN CENTER LATITUDE (de
  BHSize_:0001CB60))
```

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 nutural 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.