

Custom SI Scripting for the AD951A and AD953A

Table Of Contents

| | |
|--|----|
| Custom SI Scripting for the AD951A and AD953A..... | 5 |
| Overview..... | 5 |
| Creating Scripts with ScriptPad..... | 5 |
| Using Scripts..... | 5 |
| Contacting Tektronix..... | 6 |
| Script Language Rules and Syntax..... | 6 |
| Syntax Definition..... | 8 |
| Field Definition..... | 8 |
| field_name..... | 8 |
| format..... | 8 |
| enum..... | 9 |
| format_name..... | 10 |
| fixed_value..... | 10 |
| field_ref..... | 10 |
| calculation..... | 11 |
| validation..... | 12 |
| string fields..... | 13 |
| Loop Definition..... | 14 |
| valid_descriptors..... | 14 |
| Conditional Fields, Miscellaneous Definitions, Keywords..... | 15 |
| Conditional Fields..... | 15 |
| access_key..... | 17 |
| association..... | 18 |
| section_id..... | 19 |
| macro..... | 19 |
| group..... | 20 |
| multi_string_struct..... | 20 |
| rawbytes..... | 20 |
| Inbuilt Symbols and Functions..... | 20 |
| fThis..... | 20 |
| fNext..... | 20 |
| fDescriptor..... | 20 |
| fnBITSUM(fieldID)..... | 21 |
| fnBITLEN(fieldID)..... | 21 |
| fnFROMBCD(expression)..... | 21 |
| fnITERATIONS(fieldID)..... | 21 |
| fnSTRLEN(fieldID)..... | 21 |

Table of Contents

| | |
|--|----|
| References | 21 |
| Script Language Formal Definitions | 22 |
| Table and Descriptor Syntax | 22 |
| Support Definitions | 23 |
| Axioms | 24 |
| Index | 25 |

Custom SI Scripting for the AD951A and AD953A

This note describes how to create and use scripts. It includes all the syntax and associated rules together with examples as appropriate.

Overview

A number of MPEG Test System applications perform an initial analysis of a transport stream in order to interpret and display the System Information (SI) in a format which is easily understood by the user. Examples of applications that do this are the TS Analyser and the Multiplexer.

Custom System Information (SI) Scripting provides flexibility by allowing the user to specify SI in a simple text (script) file. Although primarily aimed at custom or private SI data carried within streams conforming to one or more broadcast standards, tables specified in the standard templates can also be redefined if required. New broadcast standards can be completely defined using scripting. Note however that, scripts specify the structure of the SI held within the stream; they do not provide the content.

Creating Scripts with ScriptPad

Scripting requires a detailed understanding of broadcast standards and C programming techniques.

The documents referenced in the last section of this note should be available.

A utility called ScriptPad is supplied to write scripts, but a simple text editor such as Microsoft Notepad may be used, if preferred. They are saved in ASCII text format by default.

There are an increasing number of proprietary standards emerging whereby analysis is allowed (possibly subject to a licence fee), but the owner of the standard does not wish for the script file to be viewed by users. This is solved by using encrypted script files combined with dongle checking. ScriptPad has an Encrypt option, such that a script file is saved in an unreadable encrypted format, that only the ScriptParser can interpret.

Using Scripts

The use of scripts is dependent upon the application that is to use them. In most cases, scripts are integrated with the application's own broadcast standard rules using purpose-provided user interface elements.

The script is checked for syntax (parsed) as part of the integration with the application. On a standard MPEG Test System installation, adding a script to an application is the only way of parsing it.

Multiple scripts may be used; thus a library of individual components may be built up and used as required. Scripts may also reference other scripts.

Contacting Tektronix

| | |
|-------------------|---|
| Phone | 1-800-833-9200* |
| Address | Tektronix, Inc. Department or name (if known) 14200 SW Karl Braun Drive P.O. Box 500 Beaverton, OR 97077 USA |
| Web site | www.tektronix.com |
| Sales support | 1-800-833-9200, select option 1* |
| Service support | 1-800-833-9200, select option 2* |
| Technical support | Email: techsupport@tektronix.com 1-800-833-9200, select option 3* 6:00 a.m. - 5:00 p.m. Pacific time |

* This phone number is toll free in North America. After office hours, please leave a voice mail message.

Outside North America, contact a Tektronix sales office or distributor; see the Tektronix web site for a list of offices.

Script Language Rules and Syntax

- Script files must have a default file extension of .scp (encrypted ones use .scx); both the .scp and .scx file extensions are automatically associated with ScriptPad as part of the installation process.
- A script file consists of a number of template and/or support definitions. Script files may also import definitions for a template(s) from another script file using the form:

```
#include "foo.scp"
```
- When including an encrypted file, refer to it using the .scp suffix. The ScriptParser will automatically search for the .scx suffix, if the file cannot be found.
- All referenced definitions must be defined before any definitions that refer to them.
- Blank lines and spaces will be ignored by the interpreter.
- Single line comments must be prefixed with //. The interpreter will ignore anything after this prefix which is on the same line. Multi-line C-style comments of the form /*...*/ are also acceptable.
- Field names are not to contain spaces. Enumerated field descriptions are quoted strings and therefore may contain spaces

The following examples illustrate the two major elements that may be specified with the script language, namely table and descriptor. The language has been deliberately designed to be as similar as possible to that used in the MPEG standards [3], with enhancements to allow the specification of additional information, such as default values, display format and validation.

Close examination of script examples will reveal that there are lines in the script that do not appear in the final table, such as the **reserved** fields. Similarly, there are elements of the displayed table that do not appear to correspond directly with the script structure. The following Syntax section describes the complete language, plus its relationship to the analysed stream.

For a formal definition of the scripting language refer to Script Language Formal Definitions.

Script Example (Program Map Table)

```

table PMT
{
  table_id 8 uimbsf eHex 0x02;
  section_syntax_id 1 bslbf eHidden 1;
  '0' 1 bslbf eHidden;
  reserved 2 bslbf eHidden 0x3 vSet;
  section_length 12 uimbsf eDec cByteLen;
  program_number 16 uimbsf eDec;
  reserved 2 bslbf eHidden 0x3 vSet;
  version 5 uimbsf;
  current_next_indicator 1 bslbf;
  section_number 8 uimbsf eDec;
  last_section_number 8 uimbsf;
  reserved 3 bslbf eHidden 0x7 vSet;
  PCR_PID 13 uimbsf eDecHex;
  reserved 4 bslbf eHidden 0xF vSet;
  program_info_length 12 uimbsf eDec cDescriptors;

  // Loop of anonymous descriptors
  loop descriptors loopen(program_info_length)

  loop elementary_streams loopen(section_length - program_info_length - 13)
  {
    stream_type 8 uimbsf eStreamType;
    reserved 3 bslbf eHidden 0x7 vSet;
    elementary_PID 13 uimbsf eDecHex;
    reserved 4 bslbf eHidden 0xF vSet;
    ES_info_length 12 uimbsf eDec cDescriptors;

    loop descriptors loopen(ES_info_length)
  }

  CRC 32 rpchof eNull;

```

Script Example (ISO_639_language_descriptor)

```
descriptor ISO_639_language_descriptor
{
  descriptor_tag 8 uimbsf eDec 0x0A;
  descriptor_length 8 uimbsf;

  loop ISO_639_language_codes looplen(descriptor_length)
  {
    ISO_639_language_code 24 bslbf eISOLatin;
    audio_type 8 bslbf eAudioType;
  }
}
```

Syntax Definition

Field Definition

The general form for field definitions within tables is as follows:

```
field_definition ::= <field_name> <bit_length> <format>
  [ (<enum>|<format_name>) <fixed_value> <validation> <default> ] ";"
```

An example:

```
table_id 8 uimbsf eHex 0x02;
```

where:

```
<field_name> = table_id
<bit_length> = 8
<format> = uimbsf
<enum> = eHex
<fixed_value> = 0x02
field delimiter = ;
```

The first three fields must be included. The remaining fields are optional, but no gaps must be left. If the last field is required, then the previous fields must contain valid data. The following notes apply to the field definition.

field_name

A field name identifies the field. Spaces are not permitted. Each section defined must include one field named 'table_id', and each descriptor defined *must* include one field named 'descriptor_tag'.

format

The data type values used here are drawn from the relevant broadcast standard specification; viz. Ref [1]. A useful type to note is iso_latin, which may be used to output strings, as in the following example where the string length is specified by the previous field.

```
text_length 8 uimbsf eDec cTextLen;
  text 8 iso_latin eNA text_length;
```

enum

Indicates the format in which the value will be displayed; viz.

- eATSCTime – h:m:s day dd mmm yyyy
- eDec – decimal value (default)
- eDecHex – decimal and hexadecimal value (e.g. PID)
- eDVBTTime – yy/mm/dd h:m:s
- eHex – hexadecimal value
- eHidden – name and value not displayed.
- eISOLatin – 8 bit ASCII characters
- eNull – value is not displayed
- eNA - don't care, use default formatting (usually eDec).
- eUnicode - 16 bit ASCII characters

The user can define a custom enum format; an example which is given below.

```
enum eStreamType
{
  0x00 "reserved",
  0x01 "MPEG-1 Video"
  0x02 "MPEG-2 Video"
  0x03 "MPEG-1 Audio"
  0x04 "MPEG-2 Audio"
  0x05 "Private_sections"
default "undefined"
}
```

The user can also use a format or enum type in place of a string value; for example:

```
enum eCountryCode
{
  900 "Scandinavia"
  901 "North America"
default eISOLatin
}
```

In this case, any value other than 900 or 901 will be converted to its three-character country code.

format_name

format_name is used in a similar manner to enums, but allows greater freedom over the formatting. It allows the user to use the value in a C style printf format string. This format string may only contain "%f" substitutions as each expression evaluates to floating point value. A format is typically used to append units to a field value and scale the value accordingly.

```
format eFrequency
{
  0.0 .. 1.0e+6] "%0.0f Hz", fThis;
  1.0e+6 .. 1.0e+9] "%5.4f MHz", fThis / 1.0e+6;
  default .. "%5.4f GHz", fThis / 1.0e+9;
}
```

Notes

1. The fThis keyword is explained in the Inbuilt Symbols and Functions section.
2. Values for a format can be expressed using scientific notation (e.g. 1.0e+6). This is to simplify the specification of the value - each value will be converted to an integer value on parsing.
3. The symbols "]" and "[" can be used to specify value-1 and value+1 respectively. This is also to simplify the script when using scientific notation.

fixed_value

A fixed_value is specified in situations where the user should not be allowed to alter it's value. The Multiplexer, for example, uses the presence of a value in this field to determine whether to lock the field in Standard Mode. In most instances only a single value will be allowed in the fixed_value field. The only exceptions are the table_id and descriptor_tag fields, thus allowing multiple identifiers to use the same template definition.

A calculation may also be used for a fixed_value. That is a fixed value based upon the values of other fields within the template. An obvious example is a descriptor length field.

Notes

1. It is an error to define both a default value and a fixed value.
2. The maximum value for a fixed value is 0xFFFFFFFF (i.e. 32 bits long). Values greater than this are truncated to this limit.

field_ref

Calculations, validations and format definitions can all refer to fields within a table, descriptor.

- Fields can be referred to using their field names. Field names should be unique *within the current scope* (see below). If an ambiguous field name is referred to the first field encountered with that name is used.
- Fields can be referred to using field references, of the form fn, where n >=0, e.g. f0, f1. If n is greater than the highest field reference an error is issued. Loops do not have a field references and they do not affect the field referencing count. Each loop or table definition has its own field reference counting. See also the section Conditional Fields.

- The scope of a field reference is the containing table or loop.
- Loops can only be referred to by name. An anonymous loop of descriptors, which does not have a name can be referred to using the pseudo field reference fDescriptors.
- fThis and fNext refer to the field for which a validation, say, is applied. If fNext is used for the last field in a loop or table definition and error is raised.

The following example illustrates these points:

```
table variable_scoping_example
{
  table_id 8 uimbsf eHex 0x43;
  samenamefield 1 bslbf; // this field is distinct...

  loop elementary_streams looplen(1) eIterations
  {
    samenamefield 4 uimbsf; // ...from this field
    loop more_streams looplen(1) eIterations
    {
      samenamefield 4 bslbf;
      if (samenamefield == 0) // refers to field above
      {
        INNERLOOP 4 uimbsf;
      }
    }
    if (samenamefield == 1) // refers to uimbsf field
    {
      CONDITION_FIELD 4 uimbsf;
    }
  }
}
```

Note that fNext is the next *valid* field. A field in a conditional expression which is not true will not match fNext.

calculation

This may be used in place of a fixed_value field (as part of a field_definition) to return a value based upon one or more field values within the template. There are several inbuilt calculations, as shown below:

- cByteLen returns sum of all bytes following the field it is used for
- cDescriptors returns byte length of next descriptors loop
- cNextByteLen returns byte length of next field
- cIterations returns the number of iterations of the following item. Next item must be a loop.
- cTextLen returns the number of characters in the following field. The following field must be a string field (iso_latin or unicode) or an error is issued.

Custom SI Scripting for the AD951A and AD953A

The calculations make good use of the inbuilt symbols and functions, as illustrated by the `cByteLen` example below. Note that all inbuilt calculations are defined in the auto-loaded script.

```
calculation cByteLen
{
  fnBITSUM(fNext) / 8
}
```

Example field definitions might be:

```
Section_length 8 uimsbf eNA cByteLen
text_length 8 uimsbf eNA cTextLen;
```

A single field name can be used in the calculation column. This is to avoid the necessity to define a calculation of the form:

```
calculation cValueOfField
{
  field_name
}
```

This might be used to ensure that a field always takes the value of another field.

validation

There are several in-built validation types:

- eNA not applicable
- vSet all bits are set
- vClear all bits are cleared
- vPid Pid value in range 0 to 8192

If the default validation type of eNA is used for any of the following reserved field names, then standard validation is automatically performed. This may be overridden by specifying a different validation type.

- table_id - $\geq 0x40$ and $\leq 0xFE$
- descriptor_tag - $\geq 0x40$ and $\leq 0xFE$
- section_syntax_id - 1
- section_length - sum of following fields
- version - < 32
- section_number - \leq last_section_number
- last_section_number - \geq section_number
- current_next_indicator - < 2
- reserved - vSet
- reserved_future_use - vSet

An example of a user defined validation for section_number is shown below:

```
validation section_number
{
  fThis <= f10    // f10 is last_section_number
}
```

If the user defined validation name is the same as one of the predefined fields in the above list the validation is applied to all fields of that name. To apply a validation to a single field (either from the above list or any other field) the following format is used:

```
validation my_section_number_validation
{
  fThis == 1
}

...

section_number 8 uimsbf eNA eNA my_section_number_validation
```

Note

String fields that use either the iso_latin or unicode format do not use the validation field. Instead the field position is used to specify the string length. Refer to string fields for further details.

string fields

For fields with the format iso_latin or unicode the following notes apply.

- A default value can be used – this must be a quoted string, e.g. "© Tektronix"
- The validation expression should be an expression, rather than a conditional expression. The result of this expression is used to validate the number of characters in a string. If the expression is a single field name the field name can be entered directly in the validation column.
- The function fnSTRLEN can be applied to a string field in order to determine the number of characters it contains.

An example of common string usage is as follows:

```
text_length 8 uimsbf eNA cTextLen;
text 8 iso_latin eNA eNA text_length;
```

Notes

The calculation cTextLen is explained in the previous discussion of calculation.

Loop Definition

A loop definition is used to specify a subset of a table, descriptor or loop definition where the field definitions are repeated a number of times.

The `loopen` part of the definition specifies the byte length of the loop. If the length is unknown, then the length may be specified as a number of loop iterations. In this instance, the keyword `eliterations` should follow the `loopen` definition.

Note that a loop definition may contain another loop definition.

There is a reserved loop_name of 'descriptors'. This is a special case that is not followed by a list of field definitions. Instead it means that a loop of any descriptor_definitions will be processed. The allowable list of descriptors is defined by `valid_descriptors`.

```
// Loop of anonymous descriptors
loop descriptors loopen(program_info_length)

loop elementary_streams loopen(section_length - program_info_length - 13)
{
  stream_type 8 uimbsf eStreamType;
  reserved 3 bslbf eHex 0x7 vSet;
  elementary_PID 13 uimbsf eDecHex 0 vPid;
  reserved 4 bslbf eHex 0xF vSet;
  ES_info_length 12 uimbsf;

  loop descriptors loopen(ES_info_length)
}
```

The validation for all loops is that the `loopen` expression when evaluated gives the total number of bytes contained within the loop. If the keyword `eliterations` (see above) is used the `loopen` expression should evaluate to the number of iterations of the loop.

Notes

1. A loop does not adjust its length when the result of its `loopen` expression changes (rather a warning will be issued). To remove the warning enough iterations must be added to the loop so that the validation succeeds.
2. A loop of anonymous descriptors can not be of type `eliterations`.

valid_descriptors

This defines a valid list of descriptors for each defined section. The descriptor list may include MPEG defined descriptors (tag range 0 to 63), as well as those defined in the script file(s). The value associated with each descriptor is the descriptor tag. If there is no `valid_descriptors` definition for a section then a warning message will be logged. The message may be suppressed by defining an empty definition.

An example for the PMT table within the ATSC standard is shown below:

```
valid_descriptors PMT
{
  2 .. 18,
  128,
  0x81,
  0x85 .. 0x87
}
```

Valid descriptors can also be built up across different script files. For example:

| | |
|---|---|
| <pre>script1.scp contains: valid_descriptors PMT { 2 .. 18, 128, 0x81 }</pre> | <pre>script2.scp contains: valid_descriptors PMT { 0x85 .. 0x87 }</pre> |
|---|---|

Subsequently the Script parser will accept that valid descriptors are:
2 .. 18, 128, 0x81, 0x85 .. 0x87.

Conditional Fields, Miscellaneous Definitions, Keywords

Conditional Fields

There are occasions where the field definition list for a template may vary depending upon the value of one or more field values. An example of conditional fields is shown below:

```
table test_table
{
  table_id 8 uimsbf eHex 0x2;

  field1 8 uimsbf eDec eNA eNA 0x1;
  if (f1 > 0)
  {
    field2 8 uimsbf;
    field3 8 uimsbf eDec 0x1;
  }
  else
  {
    field2 8 uimsbf; // ok to have same names
  }

  if (f3 != 0)
  {
    field4 8 uimsbf;
  }
  else
  {
    else
    field4 8 uimsbf;
  }
}
```

Some points to note are:

- f1, f2, etc. are field numbers; the fields are counted from the beginning of the section. In this example the field_names correspond to the field numbers – but that will not always be the case!
- If the code is analysed, some confusion may arise as to which field_name is 'f3'; it depends upon which path conditions dictate. The answer is that field numbering passes through the longest path possible, thus encountering all field_names.

From the example above:

```
if (f3 != 0)
{
  field4 8 uimsbf;
}
else
{
  else
  field4 8 uimsbf;
}
```

If the value of field1 is set to 0 the field f3 cannot be identified. The value will default to 0 and thus the else part of the condition becomes valid. Care should be taken when using a field in a condition which is itself part of a condition.

- Actual field names may be used in conditions rather than field numbers to aid clarity, if they are unique. The following is equivalent to the above example:

```
if (field3 == 0)
{
  field4 8 uimsbf;
}
```

- Care should be taken when using calculated fields in conditional expressions, e.g.

```
section_length 8 uimsbf;
if (section_length == 0)
{
  field 8 uimsbf;
}
```

can not be evaluated properly as if section_length is zero then the field should be valid, but if this field is valid the section_length evaluates to 1 !

Notes

1. Multiple nesting of conditions is allowed.
2. Each item in a condition is at the scope of its parent. This also applies to items which are nested within multiple condition statements.

access_key

This definition is used to identify those fields within a template that uniquely reference an instance. This definition is optional; if no definition is specified, the following default definitions will be used:

```

access_key <section_name>
{
  f0, f5, f7, f9 // table id + table extension + version + section_number
}

access_key <descriptor_name>
{
  f0 // descriptor tag
}

```

Notes

1) The string 'pid' can be used anywhere in the field list.

Example:

```

access_key <section_name>
{
  pid,f0,f5 // PID + table id + table extension
}

```

- 2) If the list of fields is empty the access_key evaluates to -1.
- 3) access_key is a 64 bit value. If the access_key comprises more than 64 bits, the most significant bits are lost.
- 4) Only those fields that appear before the first loop are valid for use by the access_key definition.

association

The association definition is used to specify miscellaneous characteristics of a table that are not covered elsewhere. The format specification is particularly chosen to be extensible. Generally the format of an association is shown below, where the Table id, Descriptor tag and Operand may represent multiple (range) of values:

```
<enum type> <Table id> <Operand>
<enum type> <Descriptor tag> <Operand>
```

The currently supported characteristics are:

| Enum Type | Operand | Description |
|-----------------|-------------|--|
| assocPid | Pid | Hardcodes the given Table id to the fixed Pid value. This is typically used for the DVB tables. |
| assocCycleTime | Time (ms) | Specifies the maximum repetition interval, as defined in Test 3.2 of Ref [1]. Measurements are made between the last packet of successive complete subtables. |
| assocStreamType | Stream type | This is only applicable to the PMT table. It specifies that the SI information (of Table id) is associated with the given stream type, and may be located on the elementary_PID. |
| assocDescriptor | Table field | Specifies that a 'table field' is associated with the descriptor tag. This will be useful information when processing the descriptor. |

The following example demonstrates it's usage.

```
association EIT
{
  assocPid 0x4E 0x12;
  assocPid 0x4F 0x12;
  assocPid 0x50..0x5F 0x12;
  assocPid 0x60..0x60 0x12;
  assocCycleTime 0x4E 2000;
  assocCycleTime 0x4F 10000;
  assocCycleTime 0x50..0x51 10000;
  assocCycleTime 0x52..0x59 30000;
  assocCycleTime 0x60..0x61 10000;
  assocCycleTime 0x62..0x69 30000;
}
association PMT
{
  assocStreamType 0x3B 0x0B;
}
```

section_id

This is used to customise the default section identifier string, as used, say, by the Multiplexer Navigator view. The format of the definition is very closely aligned to that used for the C language printf function. The following format will be used if there is no section_id definition for a table section:

```
"Table Id %d Extension %d Version %d Section %d [Pid %u (0x%X)]", f0, f5, f7, f9, pid, pid
```

The argument list can be either a fixed value, a field value or a reference to the pid that the table is carried on. Field values are identified as, fn (n >=0), where n is the field index in the table definition. Therefore, f7 references the seventh field, which is the version number. This default section_id makes use of the fact that all table definitions use the same structure for the first 11 fields.

macro

A commonly used section of script can be defined once and used in several places. The section of script is known as a macro.

Example

```
macro CRC { CRC_32 32 rpchof }
macro TABLEHEADER( __IDLIST__ )
{
  table_id 8 uimsbf eDec __IDLIST__ ;
  section_length 8 uimsbf;
}

table XXX
{
  TABLEHEADER(" [0x1,0x2] ")

  <other fields>

  CRC()
}
```

When the table is parsed each name which matches a macro is replaced by the text in the macro definition. Arguments to macros must be supplied in the form of a list of quoted strings. Each string is replaced wherever its corresponding argument appears in the macro text. The number of arguments supplied must match the number the macro is expecting. If a macro takes no arguments (e.g. CRC) then empty brackets must be used, as above.

group

A selection of consecutive fields may form a logical group within the context of the overall table or descriptor format. This keyword allows the fields to be referred to by a group name and indented by one level in the displayed output.

Example

```
group LLC_SNAP_Header
{
  DSAPAddress 8 uimsbf;
  SSAPAddress 8 uimsbf;
  Control     8 uimsbf;
  OUI        24 uimsbf;
  ProtocolType 8 uimsbf;
}
```

multi_string_struct

The ATSC multiple string structure is represented by this script item. The multi-string structure is atomic, i.e. all that is required is the keyword and a name. Its use results in all the multi-string structure being added to the table which is being parsed.

rawbytes

The rawbytes keyword can be used to define a block of opaque byte data. When parsed the script will read the number of bytes specified by the *length* keyword.

Example

```
rawbytes length (512)
```

If the *eSkip* keyword has been specified for the rawbytes the bytes read are discarded. If not the parsed bytes are added to the current table or descriptor but only the number of bytes read is visible. If the table or descriptor is subsequently encoded the bytes will also be written to the output buffer.

The keyword *eHidden* can be used to indicate that an application should suppress display of the rawbytes item.

Inbuilt Symbols and Functions

fThis

This symbol is replaced before an expression evaluation with the name of the actual field it is being used for, e.g. f3.

fNext

This symbol is similar to fThis except that it is replaced with the name of the next field that uses it. For example, if it is used by field3 then fNext is replaced by f4.

fDescriptor

This symbol solves the problem whereby descriptor loops cannot be referred to because they are anonymous. It effectively becomes the fieldID for the descriptor loop. Note that it is assumed that only a single descriptor loop is used at any one level.

fnBITSUM(fieldID)

This function returns the total number of bit lengths of the fields in the template from the fieldID onwards (that is, it includes the fieldID itself). It's most obvious use is in validating the section_length field.

fnBITLEN(fieldID)

This function returns the bit length of the fieldID. fieldID can be a single field or a loop name. For loops, total number of field bit lengths within the loop are returned.

fnFROMBCD(expression)

This function returns the decimal value of the BCD expression.

```
fnFROMBCD( 0x123) == 123
```

fnITERATIONS(fieldID)

This function applies when the fieldID is a loop name, and indeed returns an error if it is not. The function returns the number of iterations of a loop.

fnSTRLEN(fieldID)

This function returns the number of characters contained in a string (iso_latin or unicode) field. It is an error to call this function on any other type of field.

References

- [1] European Telecommunications Standards Institute 2001, TR 101 290: Digital Video Broadcasting (DVB); Measurement Guidelines for DVB Systems.
- [2] A/65 Program and System Information Protocol for Terrestrial Broadcast and Cable.
- [3] Information Technology – Generic encoding of moving pictures and associated audio information, ISO/IEC 13818.

Script Language Formal Definitions

Backus Naur Form is used for the formal syntax definitions that follow.

Table and Descriptor Syntax

```
script ::= { table_definition |
  descriptor_definition |
  enum_definition |
  valid_descriptors |
  validation_definition |
  calculation_definition |
  section_id |
  access_key |
  format_definition |
  association_definition |
  macro_definition |
  include_statement |
  pragma_statement
}

table_definition ::= "table" <name> "{" <items> "}"

descriptor_definition ::= "descriptor" <name> "{" <items> "}"

object_definition ::= "object" <name> "{ <items> }"

enum_definition ::= "enum" <name> "{" <enum_statement> { <enum_statement> } "}"

valid_descriptors ::= "valid_descriptors" <name> "{" [<range_value> { ",",
  <range_value> } ] "}"

validation_definition ::= "validation" <name> "{" <condition_expr> "}"

calculation_definition ::= "calculation" <name> "{" <expr> "}"

section_id ::= "section_id" <name> "{" <section_id_format> "}"

access_key ::= "access_key" <name> "{" [(<field_name> | fn){ ",", (<field_name> | fn)}] "}"

format_definition ::= "format" <name> "{" <format_statement> { <format_statement> } "}"

association_definition ::= "association" <name> "{" <association_statement>
  { <association_statement> } "}"

macro_definition ::= "macro" <name> "{" <items> "}"

include_statement ::= "#include" <name>

pragma_statement ::= "#pragma" <pragma_name> "(" <pragma_args> ")"
```

Support Definitions

```

items ::= <item> { <item> }

item ::= { bitfield | condition | loop | group | macroname | rawbytes | multistring |
  stringloop | segmentloop }

bitfield ::= <field_name> <bit_length> <format>
  [ (<enum>|<format_name>) <fixed_value> <validation> <default>];"

fixed_value ::= ("eNA" | <calculation_name> | <field_name> | <range_value> | <range_list>

format ::= (uimbsf | bslibf | rpchof | iso_latin | unicode | huffman)

validation ::= ("eNA" | "vSet" | "vClear" | "vPid" | <validation_name>)

default ::= <value>

enum_statement ::= (<range_value> | "default") "" {ascii characters} ""

format_statement ::= (<range_value> | "default") <string_format> {" "<expr>} ";"

section_id_format ::= <string_format> [ { " "<field_name> | fn | pid } ]

association_statement ::= <association_type> <range_value> <range_list>

association_type ::= ""{ascii character} ""

condition ::= "if (" <condition_expr> ")"
  {" <items> "}
  [ "else if {" <items> "} ]
  [ "else {" <items> "} ]

loop ::= "loop" ("descriptors" | <loop_name>)
"loopleft (" <expr> ")" [<length_type>] {" <items> "}

group ::= "group" <group_name> {" <items> "}

macro ::= <macroname> "(" { <quoted_string> { <quoted_string> } ")"

rawbytes ::= "rawbytes length (" <expr> ")" [ "eHidden" | "eSkip"]

multistring ::= "multi_string_struct" <name>

stringloop ::= "stringloop loopleft (" <expr> ")"

segmentloop ::= "segmentloop loopleft (" <expr> ")"

expr ::= <sub_expr> operator <sub_expr>

condition_expr ::= (<sub_expr> condition_operator <sub_expr>) | "!" <condition_expr>

sub_expr ::= [ (<argument> (operator | condition_operator) <argument>) | <argument> ]

length_type ::= (eBit_length | eIterations)

```

Custom SI Scripting for the AD951A and AD953A

```
range_value ::= ( <value> | <value> ".." <value> )

argument ::= ( <field_ref> | <function_call> | "var(" <name> ")" | <value> )

range_list ::= "[" <range_value> { "," <range_value> } "]"

field_ref ::= ( fn | <field_name> | "fThis" | "fNext" | "fDescriptors" )

function_call ::= ( "fnBITSUM(" <fieldref> ")" |
"fnBITLEN(" <fieldref> ")" |
"fnSTRLEN(" <fieldref> ")" |
"fnITERATIONS(" <fieldref> ")" |
"fnFROMBCD(" <expr> ")"
    <name> "(" <number> { <number> } ")" )

pragma_name ::= { "partition" }
pragma_args ::= <number> | <name>
```

Notes

- 1) Round brackets may be included in an expression to indicate precedence.
- 2) fn indicates a reference to the nth field, where n>=0.
- 3) segment_loop, string_loop and the format type 'huffman' are only for use internally by the parser when processing multi_string_struct. The behaviour of these items if used in any other script is not defined.
- 4) fixed_value ::= <range_value> and fixed_value ::= <range_list> can only be used for table_id and descriptor_tag fields

Axioms

```
field_name ::= {ascii character} | "'" { 0 | 1 } "'"
name ::= {ascii character}
bit_length ::= {number}
value ::= <hex_value> | <decimal_value>
hex_value ::= "0x" {hex character}
decimal_value ::= {number}
string_format ::= printf format string
operator ::= ( "+" | "-" | "/" | "*" | "|" | "&" | "<<" | ">>" | "%" )
condition_operator ::= ( "==" | "!=" | ">" | ">=" | "<" | "<=" | "&&" | "||" )
```

Note(s)

- 1) The operators used are the C language operators.
- 2) Names cannot contain spaces.

Index

A

access_key 16, 21
AD951A 5
AD953A 5
assocCycleTime 17
assocDescriptor 17
association 17
association_definition 21
association_statement 21
association_type 21
assocPid 17
assocStreamType 17
ATSC 14, 19
audio_type 6

B

Backus Naur Form 21
BCD 20
bit_length 21
bitfield 21
bslbf 6, 10, 21

C

Cable 20
calculation 11
calculation_definition 21
calculation_name 21
cByteLen 11
characteristics 17
condition_expr 21
CONDITION_FIELD 10
condition_operator 21
Conditional Fields 15
Contacting Tektronix 6
CRC 18
Creating Scripts 5
current_next_indicator 6, 12
Custom System Information 5

D

d Extension 18
d Section 18
d Version 18

decimal_value 21
Descriptor 17
Descriptor Syntax 21
descriptor_definition 21
descriptor_length 6
descriptor_name 16
descriptor_tag 6, 10, 12, 21
Digital Video Broadcasting 20
display
 rawbytes item 19
 System Information 5
display 5
display 19
DSAPAddress 19

E

eATSCTime 9
eBit_length 21
eDec 9
eDecHex 9
eDVTime 9
eFrequency 10
eHex 9
eHidden 9, 19, 21
eISOLatin 9
EIT 17
eIterations 10, 21
elementary_PID 17
encrypted 6
eNull 9
enum 9, 17, 21
enum eCountryCode 9
enum eStreamType 9
Enum Type 17
enum_definition 21
enum_statement 21
ES_info_length 6
eSkip 19, 21
eUnicode 9
F
fDescriptor 19

Index

| | | | |
|--------------------------|-----------------------|---|-------------------|
| fDescriptors..... | 21 | macro_definition..... | 21 |
| field_definition..... | 11 | macroname..... | 21 |
| field_name..... | 8, 11, 15, 21 | more_streams looplen..... | 10 |
| field_names..... | 15 | multi_string_struct..... | 19, 21 |
| field_ref..... | 10, 21 | multistring..... | 21 |
| fieldID..... | 19, 20 | my_section_number_validation..... | 12 |
| fieldref..... | 21 | O | |
| fields..... | 13 | object_definition..... | 21 |
| fixed_value..... | 10, 11, 21 | Operand..... | 17 |
| fnBITLEN..... | 20, 21 | P | |
| fnBITSUM..... | 11, 20, 21 | PCR_PID 13 uimbsf eDecHex..... | 6 |
| fNext..... | 10, 11, 19, 21 | PID..... | 9, 16, 17, 18, 21 |
| fnFROMBCD..... | 20, 21 | PMT..... | 6, 14, 17 |
| fnITERATIONS..... | 20, 21 | pragma..... | 21 |
| fnSTRLEN..... | 13, 20, 21 | pragma_args..... | 21 |
| format..... | 9 | pragma_name..... | 21 |
| format_definition..... | 21 | pragma_statement..... | 21 |
| format_name..... | 10 | printf..... | 10, 18, 21 |
| format_statement..... | 21 | Private_sections..... | 9 |
| fThis..... | 10, 12, 19, 21 | processing..... | 21 |
| function_call..... | 21 | Program Map Table..... | 6 |
| G | | program_info_length..... | 6 |
| Generic..... | 20 | program_number 16 uimbsf eDec..... | 6 |
| group..... | 19 | ProtocolType..... | 19 |
| group_name..... | 21 | purpose-provided..... | 5 |
| H | | Q | |
| Hardcodes..... | 17 | quoted_string..... | 21 |
| hex_value..... | 21 | R | |
| huffman..... | 21 | range_list..... | 21 |
| I | | range_value..... | 21 |
| IDLIST..... | 18 | rawbytes..... | 19, 21 |
| include_statement..... | 21 | rawbytes item..... | 19 |
| INNERLOOP..... | 10 | References..... | 20 |
| iso_latin..... | 9, 11, 12, 13, 20, 21 | S | |
| L | | samenamefield..... | 10 |
| last_section_number..... | 6, 12 | scp..... | 6 |
| length_type..... | 21 | Script Example..... | 6 |
| LLC_SNAP_Header..... | 19 | Script Language Formal Definitions..... | 21 |
| loop_name..... | 21 | Script Language Rules..... | 6 |
| looplen..... | 6, 21 | ScriptPad..... | 5, 6 |
| M | | ScriptParser..... | 5, 6 |
| macro..... | 18 | scx file..... | 6 |

| | | | |
|-----------------------------------|-----------------------|---------------------------------|-----------------------|
| section_id..... | 18, 21 | Technical Note | 5 |
| section_id_format | 21 | Tektronix | 6, 13 |
| section_length..... | 6, 11, 12, 15, 18, 20 | Terrestrial Broadcast | 20 |
| section_name | 16 | test_table | 15 |
| section_number | 6, 12, 16 | text_length | 9, 11, 13 |
| section_syntax_id | 6, 12 | TR 101 290 | 20 |
| segment_loop | 21 | TS Analyser | 5 |
| segmentloop looplen | 21 | U | |
| SSAPAddress | 19 | uimsbf | 6, 10, 15, 18, 19, 21 |
| Standard Mode | 10 | unicode | 11, 12, 13, 20, 21 |
| stream_type | 6 | Using Scripts..... | 5 |
| string fields..... | 13 | V | |
| string_format..... | 21 | valid_descriptors..... | 14, 21 |
| string_loop | 21 | validating section_length | 20 |
| stringloop | 21 | validation..... | 12 |
| stringloop looplen..... | 21 | validation_definition | 21 |
| sub_expr | 21 | value | 15, 20 |
| subtables | 17 | value+1 | 10 |
| Syntax Definition..... | 8 | value-1 | 10 |
| System Information..... | 5 | var | 21 |
| System Information Protocol | 20 | variable_scoping_example | 10 |
| T | | vClear..... | 12, 21 |
| Table | 17 | vPid | 21 |
| Table Id..... | 18 | vPid Pid..... | 12 |
| table_definition..... | 21 | vSet..... | 12, 21 |
| table_id | 6, 10, 12, 15, 18, 21 | W | |
| table_id' | 8 | www.tektronix.com..... | 6 |
| TABLEHEADER | 18 | | |

