DOCUMENT

SOFTWARE USER MANUAL

(SUM – Software User Manual)

FOR

MESSAGES PARSER LIBRARY Version 2.0

Prepared for:

Albatross
SkySoft-ATM
Rte de Pré-Bois 15-17
CH-1215 Geneva
Switzerland

Disposed by:

CS SOFT a.s.
K letišti 6/1019, P.O.Box 86
160 08 Praha 6
Czech Republic

Document ID	Date	Version /	Revision	Pages
MsgsParseLib_R20_SUM	**6.6.2013**	**1.0**	**-**	**21**

Distribution
Limited to contractual parties

CHANGE LIST

Version	Date	Description	Author
1.0 A	3.5.2013	Document creation	K. Hanton
1.0 B	13.5.2013	Add compilation and building steps	P. Roth, K. Hanton
1.0 C	6.6.2013	Add library usage examples	P. Roth, K. Hanton
1.0	6.6.2013	Formal review	J. Sýkora

AUTHORIZATION

Version	Review / Date	Acceptance / Date
1.0	Ivana Holaňová / 7.6.2013	Jiří Sýkora / 7.6.2013

Version	Customer / Date
1.0	

Address/: CS SOFT a.s.
K letišti 6/1019
P.O.Box 86
160 08 Praha 6
Czech Republic

Phone: +420 220 372 402
Fax: +420 220 372 402
E-mail: info@cs-soft.cz

TABLE OF CONTENTS

1 SCOPE

This document, the Software User Manual (SUM), tells how to install, use and test Computer Software Configuration Item (CSCI), software library, MsgsParseLib.

1.1 Identification

Title:	Messages Parser Library
Identification:	MsgsParseLib
Abbreviation:	MPL (mpl)
Version:	2.0
Release:	r2.0

1.2 System overview

Message Parser Library is common software library for processing messages in ICAO and ADEXP format. In such can be used in many ATC systems and tools in this area.
Software library can be used only as a part of other software libraries or applications, which call the library services. Library provides all services only via application program interface (**API**).

Provided services are decoding and building ATC area messages. Converts message from textual representation to C programming language structures (decode) and back (build). In both ways the services provide checking of message items structure and validity. Library provides only textual/syntactical processing service; there is no any semantic work, airspace knowledge or other. Operations go as deep to provide also syntactical decode and build of ICAO field 15 (route description).

Library can handle more alternatives of one type of message (message versions, also local specifications are usual), such alternatives are called message **editions**.

Library was primary implemented as part of CS SOFT Inc. FDP system. Later was modified to be independent on FDP system developing space and to be able provided under GPL licence as part of Albatross community [1] projects.

1.3 Document overview

This document describes the theory of this MsgsParseLib software library, how to use it, call services and also technical issues how to compile, install and link to user software/applications.

2 REFERENCED DOCUMENTS

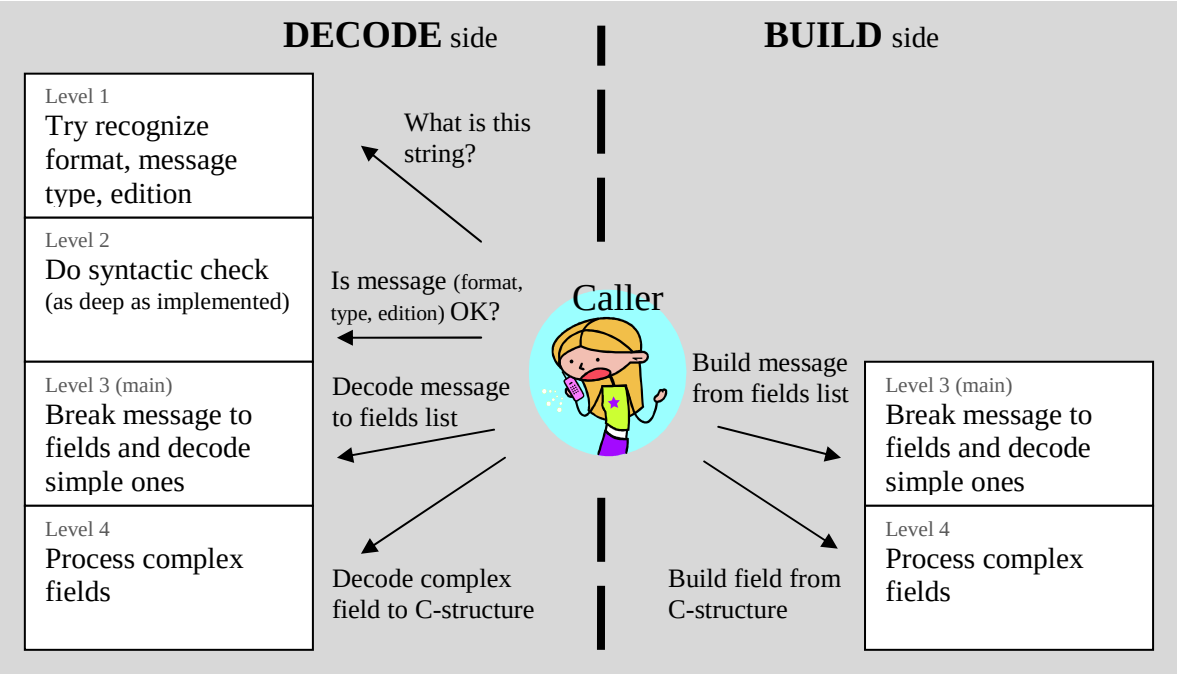
[1] Albatross, the ATM Open Source Community:
http://www.albatross.aero

3 SOFTWARE SUMMARY

This chapter contains basic description of software library, its structure and API. Also explain theory of library processing sides and levels and message editions. Explain principle of usage the meaning of each source file, their generation and message C code structures.

3.1 Software application

This software library is written in C programming language and is prepare to process ATM messages in ICAO or ADEXP format. Provides functionality to decode such message form string representation to C-structure or build back message form C-structure to string.
As software library it is not a standalone application (but tests) and it doesn't provide any user interface. It is expected to be used by other software library or application via API.



Pic: 1 – Message processing theory and API

Message processing part of software library is divided (so it's API) into two main sides Decode and Build and into 4 Levels. Sides define the direction of message processing and each level provides specific message processing services (see Pic: 1 – Message processing theory and API).

The library can handle multiple editions of one message on all four levels. Message can differ in mandatory/option fields, add specific local purpose field and field value limits; good example of message edition is old ICAO messages and FPL2012 messages.

Also provides common library API, for software library initialization, error code processing and activity logging.

Part of software library code are automatic tests, which may be compiled and run as standalone process, which runs through all prepared tests and provide result. There is infrastructure for Unit, Integration and User Requirements tests.

3.2 Software inventory

This is software library written in C language, so it must be first compiled from the source files to software library and then used in user software. Therefore software inventory has two states: source code and library.

3.2.1 Source files inventory

MsgParseLib is complex piece of software and combines more programming techniques, flex and bison tools to generate parser, XML for configuration and data structure definition and XSL transformation for generating source code based on those definitions. Therefore inventory contains various types of source files and also complex process of compilation (described later).

The source code files structure doesn't exactly follows defined parser Levels. Level 1 (message detection) service is placed directly in Level 3 (decoding) source code files and Level 2 (syntactic check) service is even implemented as part of Level 3 processing (there was no need to have Level 2 as separate service at all).

Groups of source files in `csrc/` directory:

- library API files
  - `mplDecode` – provide top access to decode functionality
    - also common library functions (initialization, configuration, …)
    - `mplConfig.h` – separated declaration of configuration API
  - `mplEncode` – provide top access to encode functionality
- fields processing (decode and encode) files (level 4)
  - `mplICAO…, mplOtherInfo` – for each ICAO field
    - `mplICAOField15` – process field as string only
  - `mplADEXPDecode`, `mplADEXPEncode` – for each ADEXP field
  - `mplRoute` – separated processing for ICAO field 15
- Filed 15 route parser source files
  - `mplRouteScanner.l`, `mplRouteParser.y` – flex and bison
- XML – data files in `xml/` directory
  - `parser.xml` – definition of messages structure
  - `parser_adexp_smart_index.xml`, `message_item_list.xml`, `parser_error_strings.xml`, `mpl_config.xml` – configuration files (below 4.2)
  - `.xsd` – XML files schema definition files
  - `.xsl` – XSL transformation for generating source code files, with message `typedefs` and support functions, from `parser.xml`
- Generated C source code files
  - `mplTypes.h`, `mplDump`, `mplNull` – from `parser.xml` via XSLT
  - `mplRouteScanner`, `mplRouteParser` – parser from flex and bison
- Testing files `test/` directory
  - `mplTest*.c` – different types and levels testing applications
  - `.xsl` – XSL transformation for Cunit result formating
- Makefiles
  - `Makefile.am` – template (for details see 4.1)

3.2.2 Library files inventory

Compiled MsgsParseLib library has much less variation in file types. At this time it is standard software library so it consists from header files and one library file (static or dynamic depends on usage).

3.3 Software environment

This software library is written in C and was developed and compiled by Linux `gcc` compiler. ICAO field 15 parser C code is generated by `flex` and `bison` lexical analyzer and parser generators. Supporting message structure functions (`dump()`, `free()`, `copy()`, …) C code is generated from XML by XSL transformation tool. There are also software compiling and building support tools.

Tools and versions list:

- Linux
- gcc version 4.5
- make version 3.81
- LIBXML2 version 2.7.7
- xsltproc
- xmllint
- yacc (GNU Bison) version 2.6.5
- flex version 2.5.37
- CUnit version 2.1 (needed for tests)
- Doxygen version 1.6.3 (needed for generate documentation)
- LogTool (optional, internal logger can be used)

3.4 Software organization and overview of operation

The software organization is divided to two main sides (Decode, Build) and four levels (see Pic: 1 – Message processing theory and API). The main API is implemented in `mplDecode` and `mplEncode` files for all formats (actually ICAO and ADEXP only). Those files contains whole first three parser levels services (recognize, check, fields list) and `mplDecode` also the library common services (initialization, configuration, …).

3.4.1 Level 4

`mplICAOField*` files implements ICAO field processing (decode/encode) level 4 functionality and the `mplADEXPDecode` and `mplADEXPEncode` the same for ADEXP fields. This way is also possible to add other message formats. Files `mplRoute` is fully separated implementation for processing ICAO filed 15 (same field is used for ADEXP messages). Implemented separation is more for technical reason, as the field 15 processing is complex task, and parser source code is generated (flex, bison).
There is also `mplICAOField15` file, but has no functionality, just process the string of field 15 to string. There is also `mplOtherInfo` files, which process ICAO field 18 as any other `mplICAOField*` files. Those are some historical level and naming inconsistencies, which should disappear in time.

3.4.2 Editions

Message editions are handled in all four level of message processing. Edition name/code is usually just underscore separated postfix text to message identification (“FPL”, “FPL_FPL2012”, “FPL_Airport”, “CFD_EFS_FPL2012” …). Edition are then handled simply via the postfix extended functions and C

code structure names on main API (`mpl_decodeFPL()`, `mpl_decodeCFD_EFS_FPL2012()`, …) and also on level 4 (`mpl_encodeICAOField10()`, `mpl_parseICAOField10_FPL2012()`, …).
Non-extended names, actually means default or standard message, but it is expected that by time all identifiers (at least on library API) will have some edition postfix.

MsgsParseLib provides service for message format, type and edition detection. Such detection is based just on basic syntax knowledge of ICAO and ADEXP format. Actually there is not implemented the edition detection. Such detection is in most cases impossible (editions differ too slightly) and none requested such service yet (editions and even types recognition is based on other system service). But the service can be fulfilled if needed.

3.4.3 Generated source code

There are two types of source code generation:

- from XML files – types definition and common support function for C code message structures. They are generated via XSL transformation.
  - `typedef char mpl_per_fpl2012_t[mpl_ARC_PER_FPL2012_LEN + 1];`
  - `char* dump_mpl_per_t();`
  - `mpl_null_mpl_message_FPL_fpl2012_s();`
- mplRoute parser – parser for ICAO field 15 is implemented in lexical analyzer `flex` and parser generator `bison`, which are then converted to pure C code.

If you don't do any changes to library source code, you don't have to care about those generations. They are processed automatically, while compiling and building the library (see 4.1.1).

3.4.4 C code structures

MsgsParseLib software library converts message to and from C code structures. Such structures are quite complex, same as are complex messages and their fields and field's items. Messages structures are generated from XML description file as described in previous section.

All ICAO and ADEXP messages are composed from set of predefined fields. Also the C code message structures are composed from predefined set of sub-structures (for each field). The sub-structure for specific ICAO field, decoded from one message may be used to build same ICAO field in other message.

Same ICAO and ADEXP (message with same meaning FPL-IFPL, …) has exactly the same C code structure. MsgsParseLib may be then used to convert ICAO messages to same ADEXP messages. (This needn't be always true, as "same meaning" messages may differ significantly by specification).

Edition of same message has usually different C code structure and often also differ in field sub-structures. Such edition structures and sub-structures names are identified by edition postfix (see 3.4.2).

3.5 Tests

Library MsgsParseLib provides wide range of automatic self-testing tools. Test tools are written in C code and use CUnit tool to administrate tests and generate their result. There are several test levels.

3.5.1 Unit tests

Tests individual library functions. Unit tests are implemented on the end of same source file as function itself. Placed inside `#ifdef` block:

```
#ifdef UNIT_TEST
…
#endif /* UNIT_TEST */
```

Unit tests launching is implement in `test/mplTestUnit.c`. Not for all functions are actually unit tests implemented.

3.5.2 Verification – Integration tests

Tests are covering whole of library functionality. Passing those test proves, that library works correct – is verified. All tests are implemented in `mplTestIntegration.c`.

3.5.3 Validation – STD Tests

Tests based on MsgsParseLib requirements. Each test validates original library requirement. Passing those test proves, that library is doing what was required – is validated. All tests are implemented in `mplTestSTD.c(h)` and their launching is in `mplTestSTDmain.c`.

Inside source file `mplTestSTD.c` are placed `doxygen` tags. This source files may be also used to generate standard software test description tables, which may be placed to some library STD documentation.

3.5.4 Working test

User working test is implemented in `mplTest.c`. This tool can process file with user filled list of messages and provide result of errors. The result briefly contains bad messages and returned decoding errors and also messages which were not same as original after building them back.

Default testing messages file is `tests/messages` of main library directory.

Working test for separated `mplRoute` works similar way (process file with ICAO field15 list) and is placed in `mplTestRoute.c`.

3.6 Contingencies and alternate states and modes of operation

N/A

3.7 Security and privacy

N/A

3.8 Assistance and problem reporting

For any MsgsParseLib related question use: servis@cs-soft.cz.

4 ACCESS TO SOFTWARE

This chapter explain software library compilation, how to include it and link it with user software and how to compile and run automatic tests and generate tests documentation. It also explains the library configuration and basic library access – initialization.

4.1 First-time user of the software

At the first time the library is in form of source code (as described in 3.2). So before can be used, must be compiled first. Even compiled library cannot be used as it is (expect the testing), but must linked and called from a user software.

4.1.1 Compile and Build

Following steps show how to compile and build MsgsParseLib with provided library Makefile.

Download & Unpack

Download msgsparse-<version>.tar.gz from ... (TODO).
Unpack the distribution it with this command:

```
$> tar -xvf msgsparse-<version>tar.gz
```

Configure

```
$> cd msgsparse-<version>
```

The default configuration is with disabled LogTools and CUnit.
For logging is used internal logger.
Can be run only test independent on CUnit (`mpl_test`, `mpl_test_route`)
Default prefix is /opt/cssoft/

For default configuration run command:

```
$> ./configure
```

Following options can be enabled:

enable LogTools	`--enable-logtools`
enable CUnit	`--enable-cunit`
change default prefix	`--prefix=<CUSTOM>`

Example:

```
./configure --enable-cunit
```

Compile

Run command:

```
$> make
```

to compile package.

Install

Run command:

```
$> make install
```

to install package.

Cleaning

To clean building directory, run command:

```
$> make clean
```

4.1.2 Compile, Build and Run tests

MsgsParseLibs contains automatic tests (as described in **Chyba! Nenalezen zdroj odkazů.**), which must be compiled and build, before running. Following steps shows how to compile, build and run MsgsParseLib Unit and Integration tests.

Download & Unpack

First see Compile and Build section above 4.1.1.

Configure

```
$> cd msgsparse-<version>
```

To Run all test must be library configured with options '`--enable-cunit`' (without can be run only mpl_test and mpl_test_route).

Compile

At default all tests are compiled.

```
$> make
```

Tests can be compiled separately with:

```
$> make csrc/test/<test_name>
```

where `<test_name>` is one of `mpl_test`, `mpl_test_integration`, `mpl_test_route`, `mpl_test_std`, `mpl_test_unit`.

Test documentation

To generate software test description tables documentation run command:

```
$> make html
```

Run test

```
$> cd csrc/test
$> ./mpl_test <test_dir>
```

where `<test_dir>` is path to test directory containing file '`messages`'. Default test directory is '`./`' .

Example messages file can be found in '`msgsparse-<versions>/tests`'.

```
$> ./mpl_test_route <route_file>
```

where `<route_file>` is file containing list of route descriptions (ICAO fields 15).

To create example `<route_file>` run command:

```
$> echo "N0467F380 KUMRU A16 LEMDA UM855 KFK UL610
TONDO/N0457F400 UM749 RUGUT UL858 OKF UM725 HDO UM748 RENDO
Q230 BEBEX T235 MITNI DCT" > <route_file>
```

Rest of tests are run without parameter.

```
$> ./mpl_test_integration
$> ./mpl_test_std
$> ./mpl_test_unit
```

Tests result

mpl_test

Each Message from test file is decoded.

If decode function end with error, message is written to file `<test_dir>/messages_bad`.
Result from decode is used to test encode part. Decoded message is encoded back and compared with original.
Differences are written in `<test_dir>/messages_cmp`.

mpl_test_integration
Result is written to XML file '`msgsparse-integration-Results.xml`'

mpl_test_unit
Result is written to XML file '`msgsparse-Results.xml`'

Rest of tests produces results to `stdout`.

For more information about CUnit visit http://cunit.sourceforge.net/documentation.html

Cleaning
To clean building directory, run command:
```
$> make clean
```

4.1.3 Link with user software

MsgsParseLib, when is build, is standard software library composed from one library binary file and group of header files (as described in 3.2.2).
The user software must first include the main header file:

```
/* include MsgsParseLib library main header file */
#include "msgsparse-1/mpl.h"
```

To link user software, the MagsParseLib library file (`libmpl-1`) must be used:

```
$> gcc <user_software_files> –lmpl-1 –o <user_software_bin>
```

4.2 Configuration

MsgsParseLib can be also configured to set common behavior of library software. Configuration is stored in XML files, each with different meaning.

4.2.1 parser_adexp_smart_index.xml

The ADEXP format is not very good communication protocol; you cannot just syntactically recognized field and subfield. One solution is a list of fields, which contains such additional information. This file has for each field information Filed or Subfield and Basic or Compound (compound - contains subfields).

4.2.2 message_item_list.xml

This file contains list of each message fields with additional information if the field is Mandatory, Optional or Ignore. MsgsParseLib use this configuration for checking missed fields.

4.2.3 parser_error_strings.xml

This file is used to define error string provided by library to user software in case of any error. Structure contains error code, symbolic code and the description strings for each library error. The recommendation is not to do any changes to those code items, because they are referenced from source code. Even the description strings are in English, so there is no possible need to do any changes to this file.

4.2.4 mpl_config.xml

This file is MsgsParseLib common configuration file. Can contains configuration parameter items in form name-value (`<item name="CHECK_ITEMS" value="NO"/>`). Actually contains some very specific parameters not any common configuration.

4.3 Initiating a session

Before software library can be used in user software, must be compiled and linked with such software (this was explained before 4.1). After this the initialization means just calling the library init function (description about `mpl_init()` in source code). Library also provide initialization state function (description about `mpl_isInitialized()` in source code).

4.4 Stopping and suspending work

As pure software library, the work is stopped along with the process. But it is good practice to call deinit function (description about mpl_deinit() is in source code).

5 PROCESSING REFERENCE GUIDE

MsgsParseLib is software library, so the best guide is good set of examples. Examples are getting from basic (initialization) to more complex library usage and on the end are also examples showing how to improve library itself.

5.1 Library initialization

```
/* include MsgsParseLib main header file library */
#include "mpl.h"

int main(const int argc, const char **argv)
{
  /* initialize */
  int result = mpl_init(argc, argv);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));
    return -1;
  }

  /* check if initialized */
  if(mpl_isInitialized() == 1)
  {
    /*
     ....
     */
  }

  /* deinitialize */
  mpl_deinit();

  return 0;
}
```

5.2 Message format and type detection

```
  /* obtain message string and declare variables */
  char message_str[MAX_BUFFER_LEN];
  strncpy( message_str, "<some message>", MAX_BUFFER_LEN );
  mpl_msg_type_e msg_type;
  mpl_format_type_e format_type;

  /* get type and format */
  mpl_getMessageType( message_str, &msg_type, &format_type);
  if (format_type == mplFormatICAO || format_type == mplFormatADEXP)
  {
    printf("error : %s\n", "unsupported format.");

  }
```

5.3 Decode Message

Parameter <type> represents corresponding message type (FPL, …).

```
  /* declare structure where decoded message is stored */
  mpl_message_<type>_s  message_<type>;
  if ( msg_type == mplMsg<type> )
```

```
{
  /* decode */
  result = mpl_decode<type>(message_str, &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }
}
```

5.4 Build message

Parameter <type> represents corresponding message type (FPL, …). Variables with same meanings are declared in previous examples.

```
if ( message_<type>.source = mplFormatICAO )
{
  /* build the ICAO format */
  result = mpl_encodeICAO_<type>(message_str, MAX_BUFFER_LEN, &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }

 }
else if ( message_<type>.source = mplFormatADEXP )
{
  /* build the ADEXP format */
  result = mpl_encodeADEXP_<type>(message_str, MAX_BUFFER_LEN, &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }

}
else
{
  printf("error : %s\n", "unsupported format.");

}
```

5.5 Build/Decode other edition (FPL 2012)

Parameter <type> represents corresponding message type (FPL, …). Variables with same meanings are declared in previous examples.

```
mpl_message_<type>_fpl2012_s  message_<type>;
if ( msg_type == mplMsg<type> )
{
  /* decode for FPL 2012 edition */
  result = mpl_decode<type>_FPL2012(message_str, &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }
}

if ( message_<type>.source = mplFormatICAO )
{
```

```
  /* build the ICAO format for FPL 2012 edition */
  result = mpl_encodeICAO_<type>_FPL2012(message_str, MAX_BUFFER_LEN,
           &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }

}
else if ( message_<type>.source = mplFormatADEXP )
{
  /* build the ADEXP format for FPL 2012 edition */
  result = mpl_encodeADEXP_<type>_FPL2012(message_str, MAX_BUFFER_LEN,
           &message_<type>);
  if (result != 0)
  {
    printf("error : %s\n", mpl_getErrorString(result));

  }

}
else
{
  printf("error : %s\n", "unsupported format.");

}
```

5.6 Decode/Build route (filed15)

```
/* obtain Field15 string and declare variables */
char route_str[MAX_BUFFER_LEN + 1];
strncpy( route_str, "<some field15>", MAX_BUFFER_LEN );
/* declare structure where decoded route is stored */
mpl_route_s route;

/* decode */
result = mpl_parseRoute(buf, &route);
if (result != 0)
{
  printf("error : %d\n", mpl_getParseErrorCode());

}

/* Build the route back as string */
route_str[0] = '\0';
result = mpl_buildRoute(route_str, MAX_BUFFER_LEN + 1, &route);
if (result == 0)
{
  printf("error : empty string builded\n");

}

/* Free the route internal structure */
mpl_freeRoute(&route, MPL_ROUTE_DONT_FREE);
```

6 NOTES

6.1 List of definitions

A	
ADEXP	ATS Data Exchange Presentation
API	Application program interface
ATM	Air Traffic Management
ATS	Air Traffic Services
B, C	
D	
E	
F	
FDP	Flight Data Processing
G	
H	
I	
ICAO	International Civil Aviation Organization
J, K, L	
M	
N	
O	
P	
Q	
R	
S	
STD	Software Test Description
T	
U	
V	

W, X, Y, Z	
XML	Extensible Marku Language
XSD	XML Schema Definition
XSL	XML Stylesheet Language

7 APPENDICES

N/A