ModBus XML Driver Generation Guide

Jibonananda Sanyal
James Nutaro

29 September 2014

Energy and Transportation Science Division

ModBus XML Driver Generation Guide

Jibonananda Sanyal
James Nutaro

Date Updated: 29 September, 2014
Date Published: 1 September, 2014

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
managed by
UT-BATTELLE, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

Page

LIST OF FIGURES

Figure **Page**

ACRONYMS

CSV	Comma Separated Value
INI	INItialization file
MDL	Modbus Definition Language
XML	eXtensible Markup Language
XSD	XML Schema Definition

ABSTRACT

This document describes utility software to facilitate automatic generation of Modbus device drivers which allows interoperable control of devices. Two software utilities are discussed: a CSV parser and a device driver generator. The CSV parser is a Python utility which translates a vendor's device register tables into an XML encoded file that describes the device and its registers. The device driver generator is C++ code that generates device drivers using the XML document created by the CSV parser.

1. OVERVIEW

The Modbus Definition Language (MDL) specification defines a uniform representation as a standard for describing Modbus device register maps. This allows automated generation of device driver software for Modbus devices. The MDL is defined by an eXtensible Markup Language (XML) Schema Definition (XSD). A specific instance of an MDL can describe individual Modbus devices using an individual XML file for each device. Please see the MDL specification for more details.

The XML files can be manually created by populating the required fields with the appropriate register/data values; however, this can be very tedious and error-prone for device vendors. To streamline the generation of these XML files, the MDL is supplied with additional utility software that enables the automated transformation of existing device register map tables to an XML files which can be used to easily generate device driver code. It is expected that the utility software should make the adoption of the new specification easier for device vendors.

The conversion from device register maps to driver code involves the following 2 step process:

Step 1: Conversion of register maps to MDL XML files. The register maps, commonly expressed in a tabular manner as a comma separated value (CSV) file, can be read into a Python program which creates an XML file adhering to the MDL.

Step 2: Generation of driver code. The MDL XML file is read by device driver generator code to create C++ source code for the driver.

The rest of the document describes the main components of the two steps: the CSV parser from step 1 and the driver code generator from step 2.

The software is organized in directories as described below:

```
ModbusXMLSchema
        .\docs
        .\schema
        .\csv-parser
        .\driver-generator
```

All descriptions in this document use the TEMPCO thermostat as an example of a ModBus device.

2. CSV PARSER

The parser reads in the device register map expressed in a tabular structure in a CSV file and outputs an MDL XML file. The Python program is supplied a command line argument specifying an INI style file. The INI file supplies all the necessary information that the parser requires for the conversion. The following sections describe how to get the software, its dependencies, and describe the INItialization (INI) file structure.

2.1 INSTALLATION

A checkout of the source code includes the necessary files for using the software from the following GitHub repository:

https://github.com/ORNL-BTRIC/ModbusXMLSchema

Use of this software requires a working installation of Python 3. The Python modules used are listed in the next section and users may have to install these modules if their Python installation does not have them.

2.2 DEPENDENCIES

The following are the Python modules used in the software. Depending on the distribution of your Python package, these modules may already be installed or may be obtained using `‘easy_install’` or `‘pip’`. The modules used are:

a. sys
b. xml
c. csv
d. difflib
e. operator
f. re
g. argparse
h. configparser
i. os
j. string

2.3 GETTING STARTED

The script runs on a command line as follows:

```
usage: csvxml.py [-h] [-i] [-v] device_ini_file

Creates an XML file from a Modbus device address map.

positional arguments:
  device_ini_file    ini-style file that provides device meta information

optional arguments:
  -h, --help         show this help message and exit
  -i, --interactive  run interactively
```

Figure 1: How to use the CSV parser script.

The script requires a device INI file which specifies the various parameters it needs for generating XML. Among many things, the INI file supplies a list of functional keywords that the software uses to search in the register table for matches. As an example, reading and writing the temperature set point for a thermostat device is fairly common, so the code looks through all instances of ‘temperature’ in a manufacturer’s spreadsheet as a possible data item that should be read or written to by a device driver. For exact matches, the script uses values in fields (such as address, length, count, etc.) to create an XML entry for the Modbus function (for details on function definitions, please see the Modbus XML specification document). However, it is impossible to have a completely consistent nomenclature across all device manufacturers. For partial matches, it calculates a similarity ratio using the Python difflib SequenceMatcher algorithm. The script may be run interactively or non-interactively. In interactive mode, the script provides a sorted listing of the closest matches and allows the user to determine which field in the manufacturer’s device spreadsheet corresponds to the field being required by the device’s XML data description. In the non-interactive mode, the script simply picks what it has algorithmically determined to be the closest match. While tests may in some cases indicate an incorrect mapping between device specifications (manufacturer’s *.csv to modbus’ *.xml), the non-interactive mode will likely require manual correction by someone familiar with the current device.

An MDL XML file for the device is generated as output.

2.4 DEVICE INI FILE

This section describes the INI style file required by the parser script.

2.4.1 Meta information

This block describes the Modbus device name and a short description of the device.

```
# An INI style file describing the inputs to the parser program. All major
# modbus device descriptions go here. Please use this as a template to provide
# the guiding values to the parser program

# Device meta information. All fields in this section are required.
[Modbus Device]
device_name             = TEMPCO Modbus device
device_description      = This is a description of the TEMPCO modbus device
```

Figure 2: Structure of a device’s meta information.

2.4.2 I/O functionality

This block provides input related to files for I/O. The ‘address_map_csv_file’ element is a required field and describes the csv filename of the Modbus device register table. The ‘lines_to_skip’ field allows the program to ignore the header lines for comments, units, and other information typically found in the top rows of a CSV file. The optional ‘output_xml_file’ field contains the desired output filename of the generated XML file. If this field is not supplied, the script derives an XML filename from the ‘address_map_csv_file’ field. The script will overwrite the output file if it exists.

```
# All I/O related functionality. All fields in this section are required.
[Input/Output]
address_map_csv_file    = test2.csv
lines_to_skip           = 1
output_xml_file         = map.xml
```

Figure 3: Flexible handling for converting a device manufacturer's CSV file to a device XML file.

2.4.3 Column Indices

Different vendors have different styles of register tables for recording the communication-specific details of their devices. Some may have a block label in the first column while others may have the register address in the first column. While these formats may be varied, the information provided is often schematically similar. To enable flexibility and ease of adoption, column indices of the different fields in the register table can be explicitly specified. This block allows users to input the corresponding column indices.

To be adaptable, the software accepts indices of the columns corresponding to the different fields, as illustrated below.

```
# Column indices (zero-based) for all fields of the address map csv file. If
# a column/field is missing, please leave it blank. Some fields are not
# optional; please consult the schema.

[Column/Field Indices]
description             = 9
addresses               = 3
length                  = 7
count                   = 4
format                  = 5
block_label             = 0
multiplier              = 6
units                   =
read_function_code      =
write_function_code     =
```

Figure 4: Flexible mapping of data columns in a device manufacturer's CSV file to specific information required to create device's XML file.

2.4.4 Additional user supplied fields

This section allows the user to specify additional columns present in the CSV file and extend the functionality of the software. Some vendors may have columns that have overlooked in this schema design. This section allows additional columns of data to be added and can be used by the parsing script. Vendors' programmers can then use the column information for any necessary changes in communication logic.

```
# User supplied column names and indices (zero-based) for additional logic that
# may be required for translating the register map to xml. Here,
# 'operation_info' is used to differentiate between R, W, and R/W addresses in
# the csv2xml.py code. See the section of the code that uses this field.

[Additional Column/Field Indices]
operation_info          = 8
```

Figure 5: Additional columns can be specified for inclusion for modifying specific parsing behavior that uses additional input.

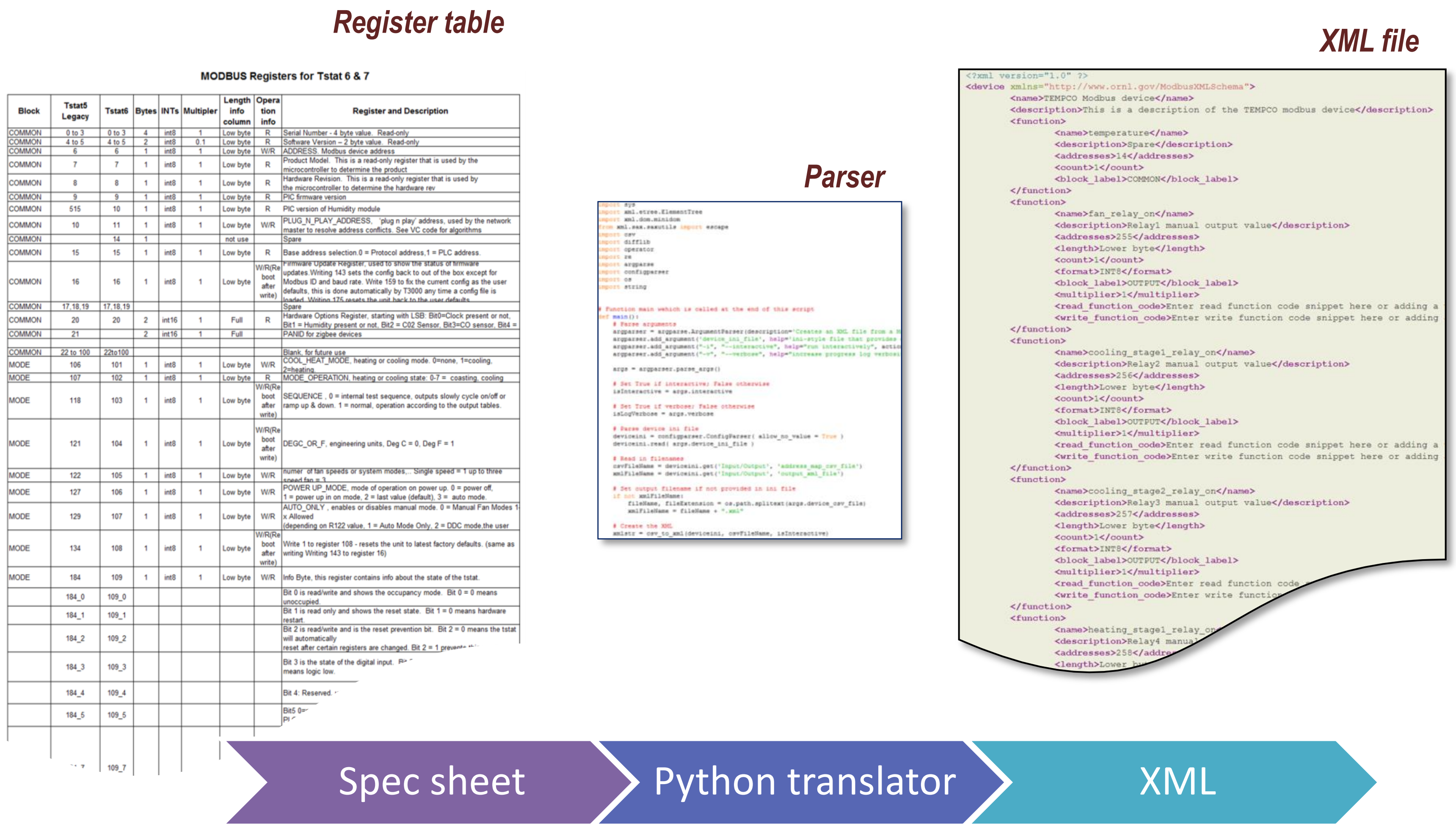
2.4.5 Function names

This section lists the functions of the Modbus device that are exposed to the device driver interface, allowing the device to be used through these functions. A list of function names and corresponding list of synonymous search terms must be provided. The synonyms are used for full and partial matches in the device driver map. The resulting MDL XML file will contain function blocks in the driver source code corresponding to the functions listed in this section. Further, when the device driver is generated from the XML, only the functions present in the XML file will have corresponding device driver code.

```
# Listing of all function names and comma separated list of searchable
# synonyms.

[Function Name Search Synonyms]
temperature             = temperature, Temperature
fan_relay_on            = Relay1 manual output
cooling_stage1_relay_on = Relay2 manual output
cooling_stage2_relay_on = Relay3 manual output
heating_stage1_relay_on = Relay4 manual output
heating_stage2_relay_on = Relay5 manual output
temperature_set_point   = Occupied setpoint
```

Figure 6: Listing of all Modbus device functions for the device which can be used to communicate with the device via the automatically generated device driver.



MODBUS Registers for Tstat 6 & 7

Block	Tstat5 Legacy	Tstat6	Bytes	INTs	Multipler	Length info column	Operation info	Register and Description
COMMON	0 to 3	0 to 3	4	int8	1	Low byte	R	Serial Number - 4 byte value. Read-only
COMMON	4 to 5	4 to 5	2	int8	0.1	Low byte	R	Software Version – 2 byte value. Read-only
COMMON	6	6	1	int8	1	Low byte	W/R	ADDRESS. Modbus device address
COMMON	7	7	1	int8	1	Low byte	R	Product Model. This is a read-only register that is used by the microcontroller to determine the product
COMMON	8	8	1	int8	1	Low byte	R	Hardware Revision. This is a read-only register that is used by the microcontroller to determine the hardware rev
COMMON	9	9	1	int8	1	Low byte	R	PIC firmware version
COMMON	515	10	1	int8	1	Low byte	R	PIC version of Humidity module
COMMON	10	11	1	int8	1	Low byte	W/R	PLUG_N_PLAY_ADDRESS, 'plug n play' address, used by the network master to resolve address conflicts. See VC code for algorithms
COMMON		14	1			not use		Spare
COMMON	15	15	1	int8	1	Low byte	R	Base address selection.0 = Protocol address,1 = PLC address.
COMMON	16	16	1	int8	1	Low byte	W/R(Reboot after write)	Firmware Update Register, used to show the status of firmware updates.Writing 143 sets the config back to out of the box except for Modbus ID and baud rate. Write 159 to fix the current config as the user defaults, this is done automatically by T3000 any time a config file is loaded. Writing 175 resets the unit back to the user defaults
COMMON	17,18,19	17,18,19						Spare
COMMON	20	20	2	int16	1	Full	R	Hardware Options Register, starting with LSB: Bit0=Clock present or not, Bit1 = Humidity present or not, Bit2 = C02 Sensor, Bit3=CO sensor, Bit4 =
COMMON	21		2	int16	1	Full		PANID for zigbee devices
COMMON	22 to 100	22to100						Blank, for future use
MODE	106	101	1	int8	1	Low byte	W/R	COOL_HEAT_MODE, heating or cooling mode. 0=none, 1=cooling, 2=heating
MODE	107	102	1	int8	1	Low byte	R	MODE_OPERATION, heating or cooling state: 0-7 = coasting, cooling
MODE	118	103	1	int8	1	Low byte	W/R(Reboot after write)	SEQUENCE , 0 = internal test sequence, outputs slowly cycle on/off or ramp up & down. 1 = normal, operation according to the output tables.
MODE	121	104	1	int8	1	Low byte	W/R(Reboot after write)	DEGC_OR_F, engineering units, Deg C = 0, Deg F = 1
MODE	122	105	1	int8	1	Low byte	W/R	numer of fan speeds or system modes,.. Single speed = 1 up to three speed fan = 3
MODE	127	106	1	int8	1	Low byte	W/R	POWER UP_MODE, mode of operation on power up. 0 = power off, 1 = power up in on mode, 2 = last value (default), 3 = auto mode.
MODE	129	107	1	int8	1	Low byte	W/R	AUTO_ONLY , enables or disables manual mode. 0 = Manual Fan Modes 1-x Allowed (depending on R122 value, 1 = Auto Mode Only, 2 = DDC mode,the user
MODE	134	108	1	int8	1	Low byte	W/R(Reboot after write)	Write 1 to register 108 - resets the unit to latest factory defaults. (same as writing Writing 143 to register 16)
MODE	184	109	1	int8	1	Low byte	W/R	Info Byte, this register contains info about the state of the tstat.
	184_0	109_0						Bit 0 is read/write and shows the occupancy mode. Bit 0 = 0 means unoccupied.
	184_1	109_1						Bit 1 is read only and shows the reset state. Bit 1 = 0 means hardware restart.
	184_2	109_2						Bit 2 is read/write and is the reset prevention bit. Bit 2 = 0 means the tstat will automatically reset after certain registers are changed. Bit 2 = 1 prevents
	184_3	109_3						Bit 3 is the state of the digital input. [illegible] means logic low.
	184_4	109_4						Bit 4: Reserved.
	184_5	109_5						Bit5 0= [illegible] Pl
	[illegible]	109_7						

```
import sys
import xml.etree.ElementTree
import xml.dom.minidom
from xml.sax.saxutils import escape
import csv
import difflib
import operator
import re
import argparse
import configparser
import os
import string


# Function main wehich is called at the end of this script
def main():
    # Parse arguments
    argparser = argparse.ArgumentParser(description='Creates an XML file from a M
    argparser.add_argument('device_ini_file', help='ini-style file that provides
    argparser.add_argument("-i", "--interactive", help="run interactively", actio
    argparser.add_argument("-v", "--verbose", help="increase progress log verbosi

    args = argparser.parse_args()

    # Set True if interactive; False otherwise
    isInteractive = args.interactive

    # Set True if verbose; False otherwise
    isLogVerbose = args.verbose

    # Parse device ini file
    deviceini = configparser.ConfigParser( allow_no_value = True )
    deviceini.read( args.device_ini_file )

    # Read in filenames
    csvFileName = deviceini.get('Input/Output', 'address_map_csv_file')
    xmlFileName = deviceini.get('Input/Output', 'output_xml_file')

    # Set output filename if not provided in ini file
    if not xmlFileName:
        fileName, fileExtension = os.path.splitext(args.device_csv_file)
        xmlFileName = fileName + ".xml"

    # Create the XML
    xmlstr = csv_to_xml(deviceini, csvFileName, isInteractive)
```

```
<?xml version="1.0" ?>
<device xmlns="http://www.ornl.gov/ModbusXMLSchema">
        <name>TEMPCO Modbus device</name>
        <description>This is a description of the TEMPCO modbus device</description>
        <function>
                <name>temperature</name>
                <description>Spare</description>
                <addresses>14</addresses>
                <count>1</count>
                <block_label>COMMON</block_label>
        </function>
        <function>
                <name>fan_relay_on</name>
                <description>Relay1 manual output value</description>
                <addresses>255</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code snippet here or adding a
                <write_function_code>Enter write function code snippet here or adding
        </function>
        <function>
                <name>cooling_stage1_relay_on</name>
                <description>Relay2 manual output value</description>
                <addresses>256</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code snippet here or adding a
                <write_function_code>Enter write function code snippet here or adding
        </function>
        <function>
                <name>cooling_stage2_relay_on</name>
                <description>Relay3 manual output value</description>
                <addresses>257</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code
                <write_function_code>Enter write functio
        </function>
        <function>
                <name>heating_stage1_relay_on
                <description>Relay4 manua
                <addresses>258</addre
                <length>Lower by
```

Figure 7: Logical representation of the CSV parser to MDL XML file

3. DEVICE DRIVER GENERATOR

This section describes the dependencies and method to compile the C++ code for generating device drivers from the XML file generated by the parser.

3.1 INSTALLATION AND DEPENDENCIES

A checkout of the source code includes the necessary files for compiling and using the software. This is C/C++ code with several external dependencies. A simple Makefile is supplied which should make the compilation straightforward.

The following are the library dependencies:

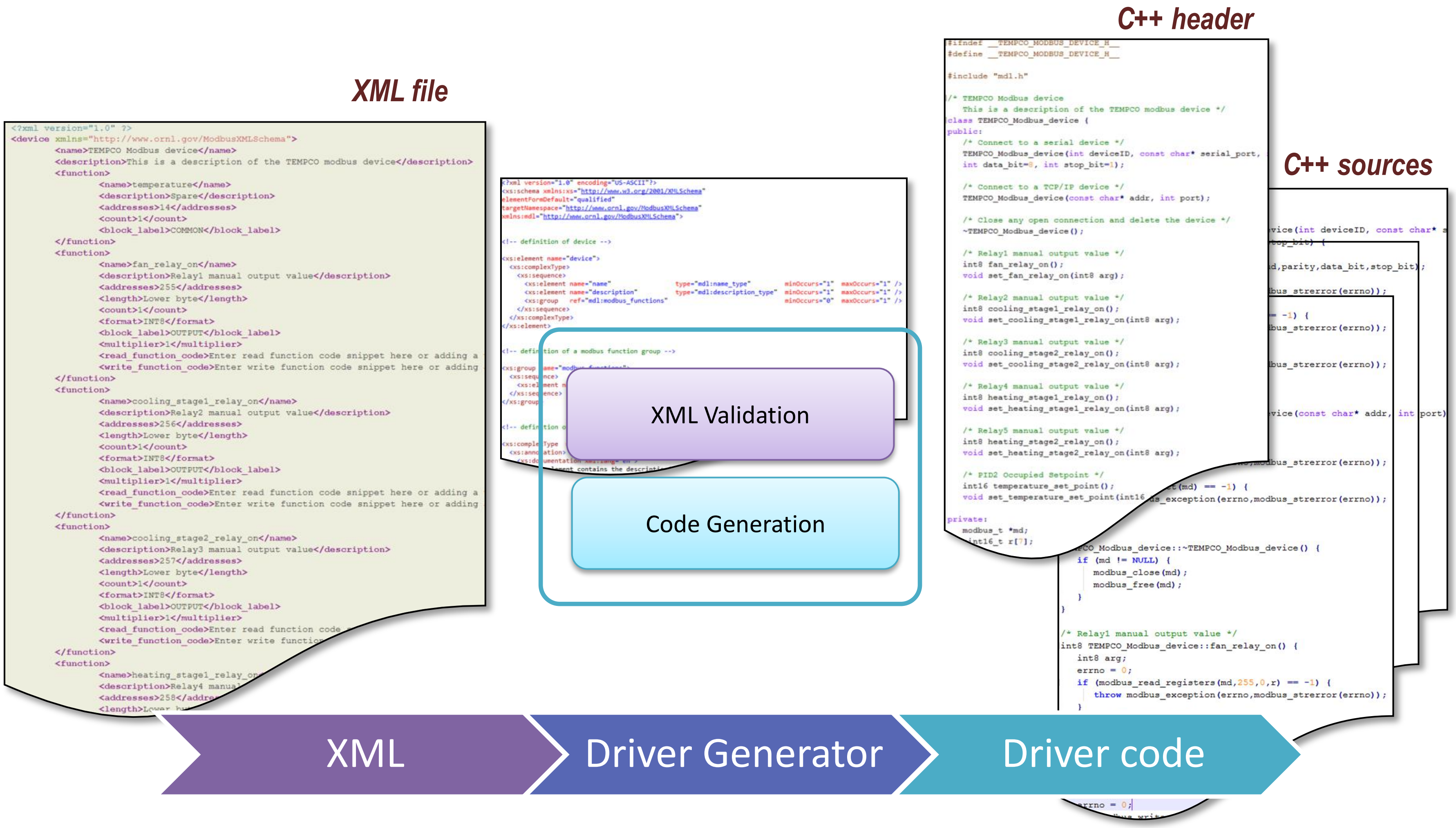
a. libmodbus
b. libxml++ 2.6
c. stl

An executable named ‘mdl2code’ is created after compiling the source code.

3.2 DEVICE DRIVER GENERATION

The executable ‘mdl2code’ accepts one argument which is the MDL XML filename (which can be created/edited manually or can be generated using the CSV parser). The executable creates <device_name>.h and <device_name>.cc files which are the corresponding device driver files for the supplied MDL XML. Spaces, if any, in the device name are replaced with underscores to create the filenames of the generated code

.



```xml
<?xml version="1.0" ?>
<device xmlns="http://www.ornl.gov/ModbusXMLSchema">
        <name>TEMPCO Modbus device</name>
        <description>This is a description of the TEMPCO modbus device</description>
        <function>
                <name>temperature</name>
                <description>Spare</description>
                <addresses>14</addresses>
                <count>1</count>
                <block_label>COMMON</block_label>
        </function>
        <function>
                <name>fan_relay_on</name>
                <description>Relay1 manual output value</description>
                <addresses>255</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code snippet here or adding a
                <write_function_code>Enter write function code snippet here or adding
        </function>
        <function>
                <name>cooling_stage1_relay_on</name>
                <description>Relay2 manual output value</description>
                <addresses>256</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code snippet here or adding a
                <write_function_code>Enter write function code snippet here or adding
        </function>
        <function>
                <name>cooling_stage2_relay_on</name>
                <description>Relay3 manual output value</description>
                <addresses>257</addresses>
                <length>Lower byte</length>
                <count>1</count>
                <format>INT8</format>
                <block_label>OUTPUT</block_label>
                <multiplier>1</multiplier>
                <read_function_code>Enter read function code
                <write_function_code>Enter write functio
        </function>
        <function>
                <name>heating_stage1_relay_on
                <description>Relay4 manua
                <addresses>258</addre
                <length>Lower by
```

```xml
<?xml version="1.0" encoding="US-ASCII"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
targetNamespace="http://www.ornl.gov/ModbusXMLSchema"
xmlns:mdl="http://www.ornl.gov/ModbusXMLSchema">


<!-- definition of device -->

<xs:element name="device">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name"           type="mdl:name_type"         minOccurs="1"  maxOccurs="1" />
      <xs:element name="description"    type="mdl:description_type"  minOccurs="1"  maxOccurs="1" />
      <xs:group   ref="mdl:modbus_functions"                         minOccurs="0"  maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:element>


<!-- defin tion of a modbus function group -->

<xs:group
  <xs:sequ nce>
    <xs:el ment
  </xs:seq ence>
</xs:group


<!-- defin tion

<xs:comple Type
  <xs:anno ation>
    <xs:do umentation
      ement contains the descripti
```

```cpp
#ifndef __TEMPCO_MODBUS_DEVICE_H__
#define __TEMPCO_MODBUS_DEVICE_H__

#include "mdl.h"

/* TEMPCO Modbus device
   This is a description of the TEMPCO modbus device */
class TEMPCO_Modbus_device {
public:
   /* Connect to a serial device */
   TEMPCO_Modbus_device(int deviceID, const char* serial_port,
   int data_bit=8, int stop_bit=1);

   /* Connect to a TCP/IP device */
   TEMPCO_Modbus_device(const char* addr, int port);

   /* Close any open connection and delete the device */
   ~TEMPCO_Modbus_device();

   /* Relay1 manual output value */
   int8 fan_relay_on();
   void set_fan_relay_on(int8 arg);

   /* Relay2 manual output value */
   int8 cooling_stage1_relay_on();
   void set_cooling_stage1_relay_on(int8 arg);

   /* Relay3 manual output value */
   int8 cooling_stage2_relay_on();
   void set_cooling_stage2_relay_on(int8 arg);

   /* Relay4 manual output value */
   int8 heating_stage1_relay_on();
   void set_heating_stage1_relay_on(int8 arg);

   /* Relay5 manual output value */
   int8 heating_stage2_relay_on();
   void set_heating_stage2_relay_on(int8 arg);

   /* PID2 Occupied Setpoint */
   int16 temperature_set_point();
   void set_temperature_set_point(int16

private:
   modbus_t *md;
   int16_t r[7];
```

```cpp
vice(int deviceID, const char* s
top_bit) {

d,parity,data_bit,stop_bit);

bus_strerror(errno));

= -1) {
bus_strerror(errno));

bus_strerror(errno));

vice(const char* addr, int port)

modbus_strerror(errno));
t(md) == -1) {
s_exception(errno,modbus_strerror(errno));


CO_Modbus_device::~TEMPCO_Modbus_device() {
   if (md != NULL) {
      modbus_close(md);
      modbus_free(md);
   }
}

/* Relay1 manual output value */
int8 TEMPCO_Modbus_device::fan_relay_on() {
   int8 arg;
   errno = 0;
   if (modbus_read_registers(md,255,0,r) == -1) {
      throw modbus_exception(errno,modbus_strerror(errno));
   }

   errno = 0;
```

Figure 8: Logical representation of the device driver is validated and used to generate driver code for communicating with the device.

4. LIMITATIONS

Known limitations of the software include:

CSV parser

a. Developed and tested on Python 3 and untested on Python 2.7.
b. Developed and tested on Linux Ubuntu platform only.
c. Support US-ASCII csv files only.

Device-driver generator

a. Developed, compiled and tested on Linux Ubuntu only.

5. LICENCE