Wolframe documentation

October 31, 2013

Table of Contents

Wolframe Application Server

Application Building Manual

Wolframe Application Server: Application Building Manual

Publication date October 31, 2013 version 0.0.3

Table of Contents

List of Figures

Foreword

The Wolframe project was started in 2010. The goal was to create a platform for fully customizable business applications that can be hosted in modern system environments.

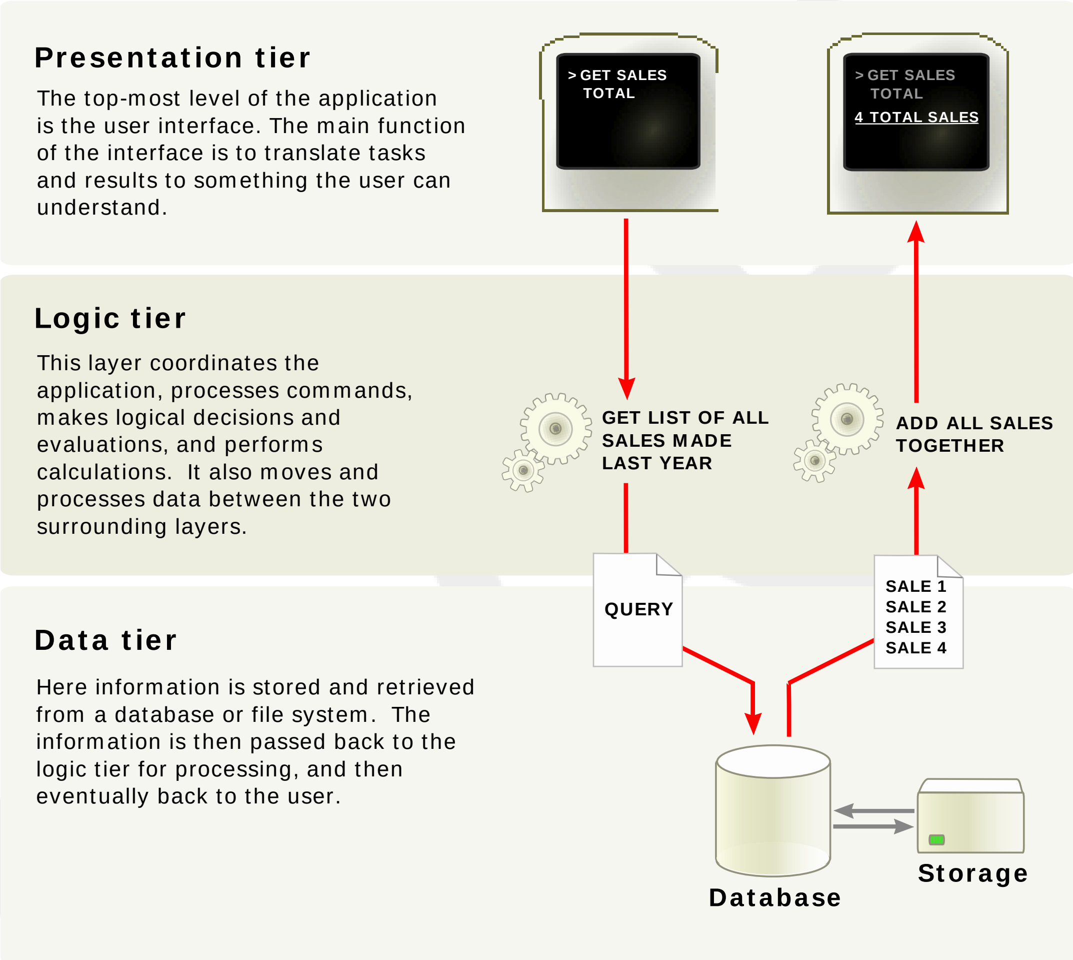
This manual introduces the architecture of Wolframe and explains how to build client/server applications with it. After reading this you should be able to create an application on your own.

Chapter 1. Introduction

1.1. Architecture

Wolframe is a 3-tier application server.

Figure 1.1. Overview



1.1.1. Presentation tier

The presentation tier of Wolframe is implemented as a thin client. It maps the presentation of the application from the request answers it gets from the server. Also the data describing this mapping is loaded from the server when connecting to it. So the whole application is driven by the server. A special use case are user interface designers that upload the presentation data for other users to the server.

1.1.2. Logic tier

The logic tier of Wolframe describes the transformation of input of the presentation tier to a set of instructions for the data tier. The input to the logic tier consists of a command name plus a structured content also referred to as document. The logic tier returns a single document to the presentation tier. The logic tier supports scripting languages to define the input/output mapping between the layers. Wolframe introduces three concepts as building blocks of the logic tier:

- *Filters*: Filters are transforming serialized input data (XML,JSON,CSV,etc.) to a unified serialization of hierarchically structured data and to serialize any form of processed data for output. Filters are implemented as loadable modules (e.g. XML filter based on libxml2, JSON filter based on cJSON) or as scripts based on a filter module (XSLT filter script for rewriting input or output)
- *Forms*: Forms are data structures defined in a data definition language (DDL). Forms are used to validate and normalize input (XML validation, token normalization, structure definition). The recommended definition of a command in the logic tier has a form to validate its input and a form to validate its output before returning it to the caller.
- *Functions*: Functions delegate processing to the data tier (transactions) or they are simple data transformations or they serve as interface to integrate with other environments (e.g. .NET). Functions have a unique name and are called with a structure as argument and a structure as result. Functions can call other functions for delegation, e.g. a transaction definition can call a .NET function for preprocessing its input or a .NET function can call a Python function to do parts of the processing.

You find a detailed description of the Logic tier and how to use it in the SDK manual.

1.1.3. Data tier

The data tier of Wolframe defines the functions for calling a transaction. The main transaction function gets a complete description of the transaction to execute as input and returns all results of the transaction as output. The logic tier builds the result data structure out of this result. The main transaction function is stateless and an abstraction of the transactional context. (The transaction context does not exist outside this function. Differently explained: Two functions do not refer to the same transaction).

Index

Wolframe Clients

Clients to access Wolframe

Wolframe Clients: Clients to access Wolframe

Publication date October 31, 2013 version 0.0.3

Table of Contents

Chapter 1. Introduction

This part of the manual describes how the user interface part (presentation tier, also called client) of Wolframe applications can be built.

A Wolframe client can be of various kinds. They all communicate with the server over a text based protocol in a plain or encrypted session. All methods used are based on open standards.

We will introduce two examples of clients: The Wolframe standard client and a web client communicating via a web server with the Wolframe application server.

After reading this chapter you should be able to create a Wolframe client based of one of these two examples on your own.

Chapter 2. Presentation Tier

This chapter describes the presentation tier and how a Wolframe user interface is built.

2.1. Wolframe Standard Client (wolfclient)

2.1.1. Architecture

The Wolframe standard client wolfclient is a thin client which executes XML requests via the Wolframe protocol and presents XML answers. It is written in Qt and is cross-platform. Qt is currently available on http://doc.qt.digia.com/qt/index.html. User interfaces for wolfclient are defined as a set of forms using standard Qt widgets and are if ever possible defined using the Qt Interface Designer (see http://qt-project.org/doc/qt-4.8/designer-manual.html).

2.1.2. Artifacts

The wolfclient renders user interface forms dynamically, this means no code generation or compilation is involved when creating user interfaces for Wolframe.

UI forms

The UI files follow the schema 'qt-ui-4.7.xsd', as documented in http://doc.qt.digia.com/qt/designer-ui-file-format.html. The UI files have the extension *.ui*

UI files are created and edited with the Qt designer.

UI form translations

The wolfclient uses the Qt translation format, version 2.0 for form translations as described in http://doc.qt.digia.com/qt/linguist-ts-file-format.html. Those are the files with extension *.ts*.

The translation files can get merged and generated with the *lupdate* tool, then translated with the *Qt Linguist*.

The Qt client needs the files in compiled form as files with the extension *.qm*. The *lupdate* tool is taking care of that.

Read more on translations in http://doc.qt.digia.com/qt/linguist-manual.html.

Resources

Binary resource files contain images for the user interface.

Binary resource files (extension *.rss*) are compiled from a XML file (extension *.qrc*) with the *rcc* resource compiler.

2.1.3. Programming the Interface

Programming means we annotate the XML of the UI form files with some extra properties. They control the following things:

- Which events in the current form replace it with a new form, e. g. clicking the *Edit* button loads the form called *edit_item*.
- When and how requests to the Wolframe server should be sent and how the results should be interpreted when adding data to the widgets, e.g. executing a *save item request* with all the data in the text fields of the form added to the request XML.

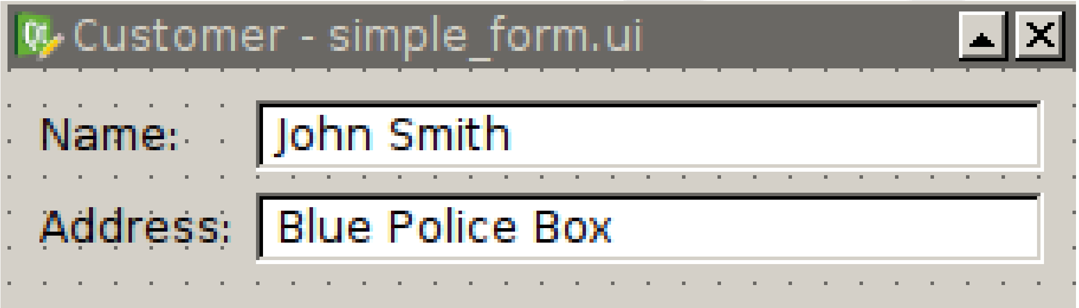
Mapping XML Data

Starting Position

For mapping data structures from the user interface elements to the data description needed to fulfil an interface for a server request we need some kind of translation. An implicit mapping would only be able to describe very trivial data mappings. After drawing the user interface this translation has to be defined. On the other hand the requests answer returned by the server has to be mapped to be shown in the user interface elements view. Here applies the same: Some kind of translation is needed to map a server data structure to the user interface elements.

First Example

Lets have a look at a `QLineEdit` element of a form and a possible XML representation of the data used for a request.




```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!DOCTYPE customer SYSTEM 'Customer'>
<customer>
  <name>John Smith</name>
  <address>Blue Police Box</address>
</customer>
```

For an insert or update request that transmits all data of the form to the server we have to fill the name field and the address field into the request data structure XML. The translation is defined as dynamic property "action" or "action." plus a suffix for the action identifier if needed. We will explain this naming of actions later. The value of the property is describing the request and could look as follows:

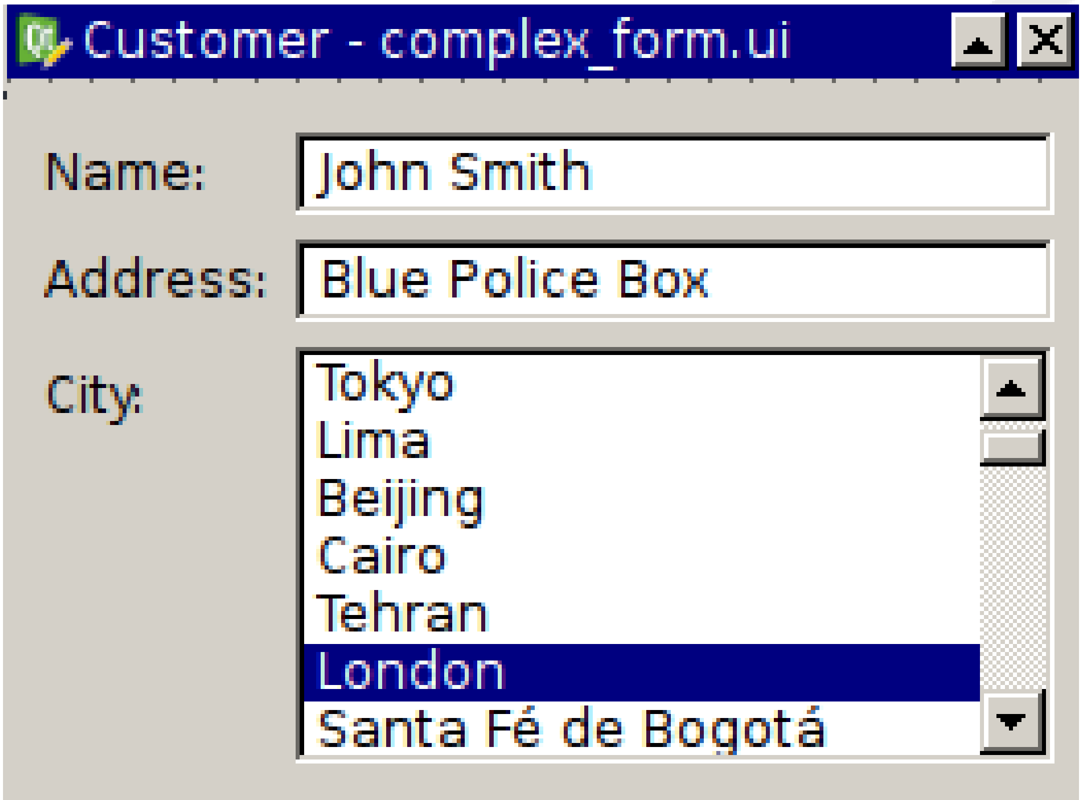
```
update: Customer customer {name{{main.name}}; address{{main.address}}}
```

For the initial filling of the form with data we submit a request that just sends an id to the server. The answer that is returned by the server has then to be translated to fill the name field and the address field of the form. The translation is defined as dynamic property "answer" or "answer." plus a suffix for the action identifier. As in the request example we postpone a detailed explanation of this action naming semantics. The value of the property is describing the answer in the same language as a request and could look as follows:

```
Customer customer {name{{main.name}}; address{{main.address}}}
```

Another Example

Some elements are more complicated than that. They present the user a list of options or items the user to pick from, e.g. a list of cities.



When the form is saved, the currently selected element is written into the resulting XML:

```
<customer>
  <name>John Smith</name>
  <address>Blue Police Box</address>
  <city>6</city>
</customer>
```

In this case the widget with the city list can load its own domain data as a separate XML request:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE city SYSTEM 'CityListRequest'>
<cities/>
```

and the corresponding domain load request answer definition in the dynamic property "answer" could look like follows:

```
    CityList cities {city[] {id={main.city.id}; {main.city.value}}}
```

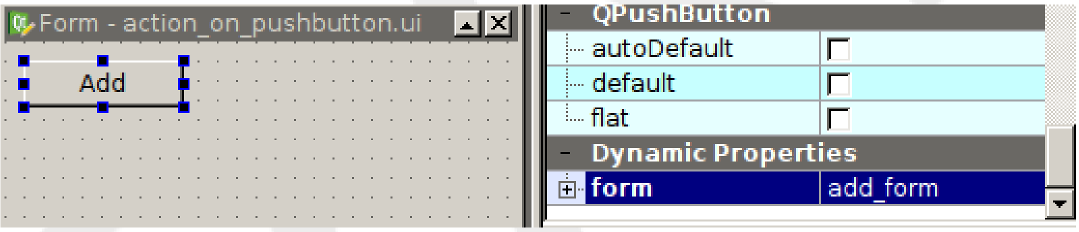
The answer contains all possible values in the domain, in our case a list of all cities and their internal id.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE "cities" SYSTEM "CityList">
<cities>
<city id='1'>Tokyo</city>
<city id='2'>Lima</city>
<city id='3'>Beijing</city>
<city id='4'>Cairo</city>
<city id='5'>Tehran</city>
<city id='6'>London</city>
</cities>
```

Switching UI forms

A UI form contains a set of widgets, the dynamic property `form` contains the name of a widget (without extension *.ui*) to load.

For linking a push `QPushButton` click in the Qt designer to the switching of the form you have to attach a dynamic property named `form` of type `string` to the corresponding widget of type `QPushButton`:



Before loading the next form the client terminates all current requests, for instance a save request of the form data. In case of an error in an action any defined switching of the form is cancelled.

States and properties of widgets

Some properties are reserved for states steering the behaviour of the user interface:

- `initialFocus`: This is a thing which got forgotten in the Qt design. We can set a boolean value to one widget in a form which should get the initial keyboard focus.

Widget properties as dynamic property values

Dynamic properties can reference properties of widgets like for example `property = {variable expression}`.

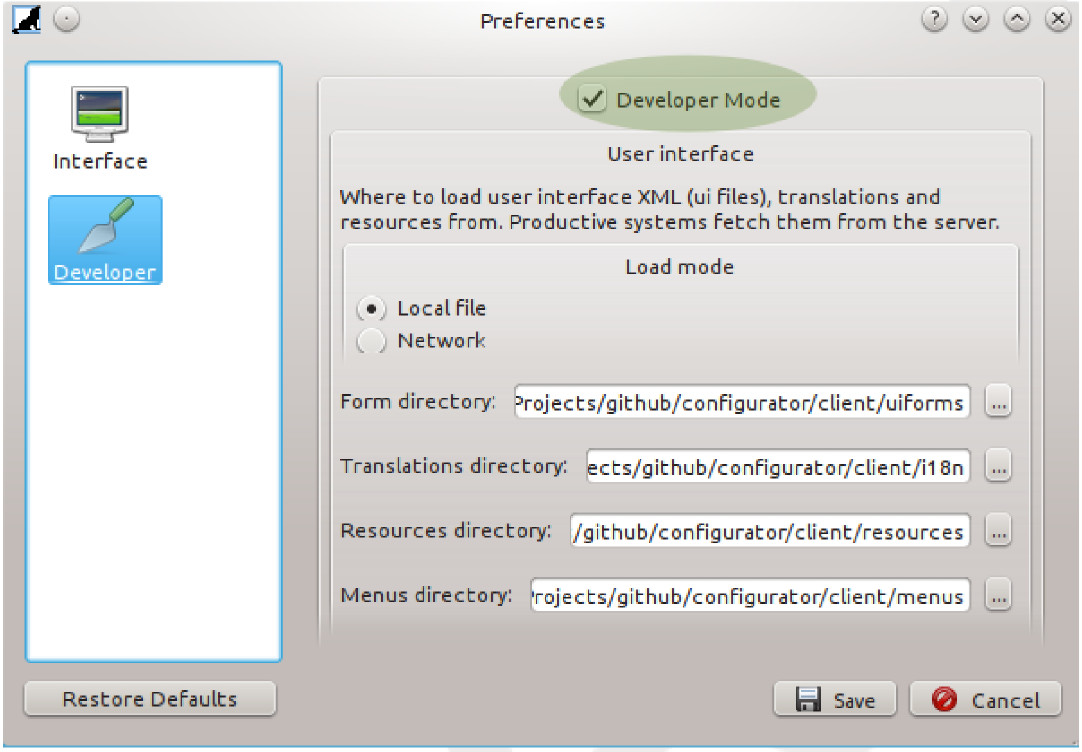
The expression can reference addressable widgets and their properties. Every Qt class has its very own set of properties it understands.

2.1.4. Eliminating Interface Defects

Functional defects in the user interface like for example syntax errors in the definitions of the request answer can be eliminated by inspecting the error messages reported by the wolfclient in developer mode and fixing the interface accordingly.

Switch the Developer Mode On

In order to inspect the internals of your client program, we have first to switch on "Developer Mode" in the "Developer" context of the "Preferences Dialog". The following picture emphasizes the check box you have to enable (highlighted green).

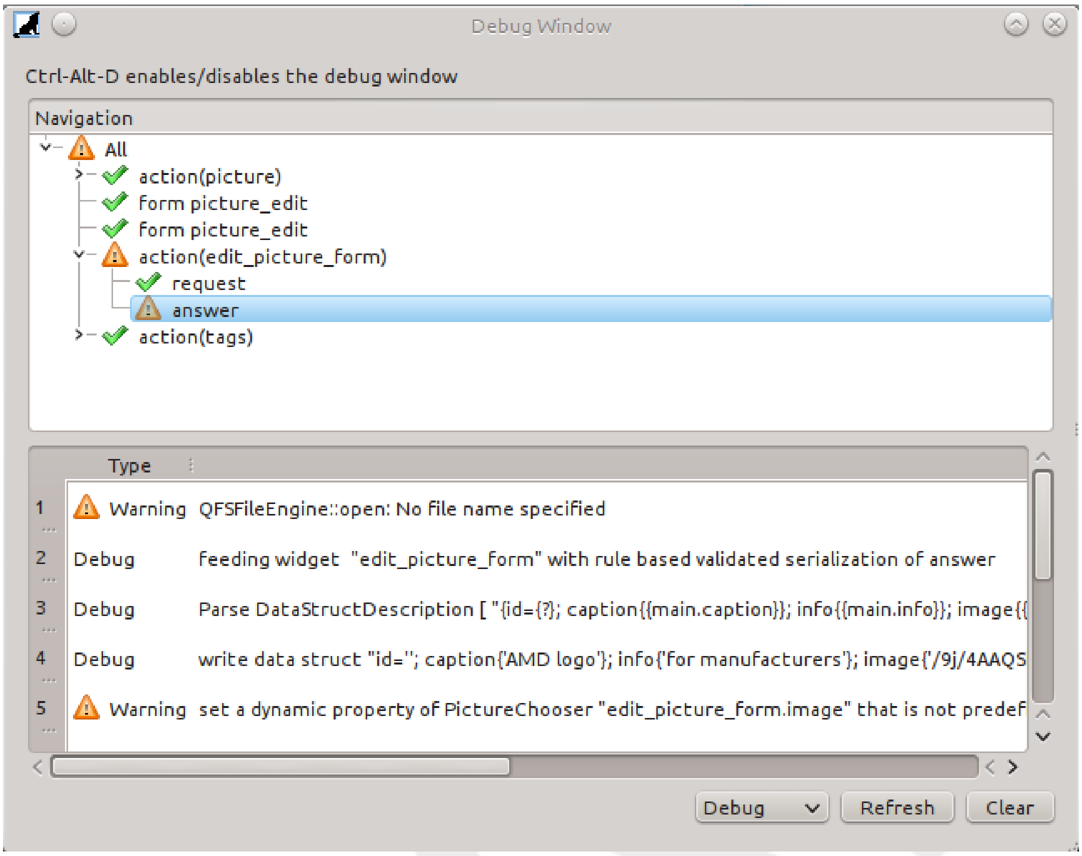


Inspect Errors and Warnings and Debug Messages Reported

To inspect internal messages reported by the wolfclient in developer mode we have to open the debug window. The debug window is opened by clicking on the bug icon in the main tool bar or via the developer context menu. The following picture shows an example debug output. Each action we do from now on with the debug window opened can be followed on the level of messages it emits.

We can see the messages in the message list when clicking on the "Refresh" button. The navigation allows us to restrict our focus on messages on a node in the object tree by clicking on it. Clicking on the root node shows all messages in the recent history. The history starts with the last main node created before opening the debug window. All message restrictions show the messages in order of their emission. We can restrict also on the severity of messages in the severity level selection (the select box set to "Debug" as default left of the "Refresh" button).

The "Clear" button allows us to empty the recent history without closing the debug window.


Debug Window
Ctrl-Alt-D enables/disables the debug window
Navigation
All
action(picture)
form picture_edit
form picture_edit
action(edit_picture_form)
request
answer
action(tags)
Type
1 Warning QFSFileEngine::open: No file name specified
2 Debug feeding widget "edit_picture_form" with rule based validated serialization of answer
3 Debug Parse DataStructDescription ["{id={?}; caption{{main.caption}}; info{{main.info}}; image{{
4 Debug write data struct "id=''; caption{'AMD logo'}; info{'for manufacturers'}; image{'/9j/4AAQS
5 Warning set a dynamic property of PictureChooser "edit_picture_form.image" that is not predef
Debug
Refresh
Clear

Index

Building Wolframe Services

Service Development Kit

Building Wolframe Services: Service Development Kit

Publication date October 31, 2013 version 0.0.3

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

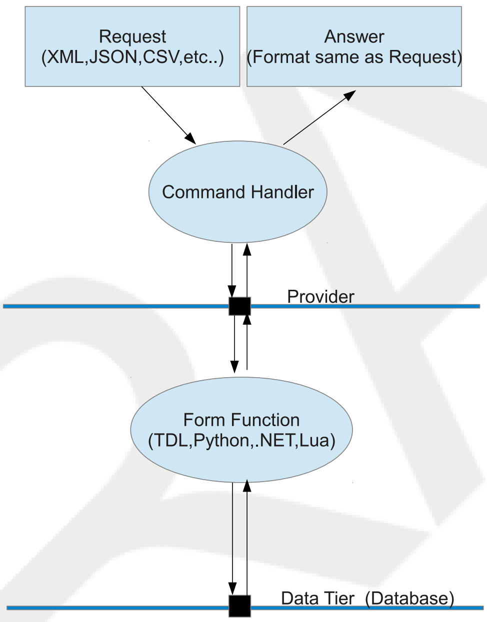
1.1. Introduction to Service Development

Wolframe application server requests consist of a named command and a structured content (document) as argument.

In this chapter we describe how Wolframe application server requests are processed. You will understand the involved bricks of software and how they are configured to build the backend of your Wolframe application.

The following illustration shows the processing of one client request to the server. A call of the Wolframe logic tier gets to a command handler that calls functions given by the provider to perform the transaction requested.

Figure 1.1. Overview



We will describe now the Wolframe standard command handler and how it is configured. Then we will show how to write programs that declare the functions executing the requests and how you link them to your application.

Chapter 2. Command Handler

This chapter introduces the standard command handler of the logic tier.

2.1. The Standard Command Handler

2.1.1. Introduction

The Wolframe standard command handler is called *directmap* and named so in the configuration because it only declares a redirection of the commands to functions based on the document type and the command identifier specified by the client in the request.

The declarations of the Wolframe Standard Command Handler (*directmap*) are specified in a program source file with the extension '.dmap' that is declared in the configuration.

2.1.2. Example Configuration

The following configuration declares a program `example.tdl` written in the transaction definition language (TDL) to contain the function declarations for the provider that can be called by the command handler. It declares the database with name pgdb to be used as the database for transactions. It loads a description `example.dmap` that will declare the mappings of commands to the filters used and functions called. It specifies the filter with name libxml2 to be used for documents of format XML and the filter with name cjson to be used for documents of format JSON, if not specified else in `example.dmap`.

```
  ; Simple Data Processing Configuration Example
Processor
{
     ; Programs to load:
     program example.tdl              ; a program with functions (in TDL)
     database pgdb                    ; references transaction database

     ; Command handlers to load:
     cmdhandler
     {
          directmap                   ; the standard command handler
          {
               program example.dmap   ; description of command mappings

               filter XML=libxml2     ; std filter for XML document format
               filter JSON=cjson      ; std filter for JSON document format
          }
     }
}
```

2.1.3. Example Command Description

The following source example could be one of the `example.dmap` in the configuration example introduced above. It defines two commands. The first one links a command "insert" with document type "Customer" as content to a transaction function "doInsertCustomer". The content is validated automatically against a form named "Customer" if not explicitly defined else. The command has no result except that its succeeds or fails. The second example command links a command "get" with a document type "Employee" to a function "doSelectEmployee". The input is not validated and the transaction output is validated and mapped through the form "Employee".

```
COMMAND insert Customer CALL doInsertCustomer;
COMMAND get Employee PASS CALL doSelectEmployee( xml ) RETURN Employee;
```

2.1.4. Command Description Language

A command map description file like our example shown consists of instructions started with `COMMAND` and terminated by semicolon ';'. The first argument after `COMMAND` is the name of the command followed by the name of the document type of the input document. The name of the command is optional. If not specified the first argument after `COMMAND` names the input document type.

Simple Document Map

The following example shows the simplest possible declaration. It states that documents with the document type "Document" are forwarded to a function with the same name "Document".

```
COMMAND Document;
```

Command with Action Prefix

The next example adds a action name to the declaration. The implicit name of the function called is insertDocument:

```
COMMAND insert Document;
```

Explicit Function Name Declaration

For declaring the function called explicitly like for example a function `doInsertDocument` we need to declare it with `CALL <functionname>`:

```
COMMAND insert Document CALL doInsertDocument;
```

Returned Document Declaration

The document type returned is specified with `RETURN <doctype>`:

```
COMMAND process Document RETURN Document;
```

or with explicit naming of a function called

```
COMMAND process Document CALL doProcessDocument RETURN Document;
```

Skipping Document Validation

If you want to skip the input document validation, either because you are dealing with legacy software where a strict definition of a schema is not possible or because the function called has strict typing and validates the input on its own (.NET,C++), then you can add a declaration `SKIP`:

```
COMMAND process Document SKIP CALL doProcessDocument RETURN Document;
```

The same you can specify for the output in the `RETURN` declaration:

```
COMMAND process Document CALL doProcessDocument RETURN SKIP Document;
```

Explicit Filter Definitions for a Command

For most processing it's enough to declare the standard filters in the configuration of the command handler. But in certain cases you want to explicitly declare a filter for a command, for example to preprocess a certain document type with an XSLT filter. Explicitly declared filters always refer to a document format and documents of other formats have to be converted first or they cannot be preprocessed. The conversions mechanisms we will explain in detail later. Explicit filter declarations are done with

- `FILTER <name>` or
- `FILTER INPUT <inputfiltername>` or
- `FILTER OUTPUT <outputfiltername>` or
- `FILTER INPUT <inputfiltername> OUTPUT <outputfiltername>`

Here is an example:

```
COMMAND process Document FILTER INPUT myXsltInputFilter
    CALL doProcessDocument RETURN Document;
```

Using Brackets

For better readability you can use optional '(' ')' brackets on the arguments. This way you can distinguish better between keywords and arguments:

```
COMMAND ( process Document )
    FILTER INPUT ( myXsltInputFilter ) CALL ( doProcessDocument )
    RETURN ( Document );
```

Overview

Each command declaration has as already explained the form

- COMMAND <doctype> [OPTIONS] ; or
- COMMAND <action> <doctype> [OPTIONS] ;

The following table shows an overview of the elements that can be used in the [OPTIONS] part of the command:

Table 2.1. Options

Keywords	Arguments	Description
CALL	Function Name	Names the function to be called for processing the request
RETURN	Document Type	Specifies the type of the document returned and forces validation of the output
RETURN SKIP	Document Type	Specifies the type of the document returned but skips validation of the output
SKIP	(no arguments)	Specifies the input document validation to be skipped
FILTER INPUT	Filter Name	Specifies that the filter <Name> should be used as input filter
FILTER OUTPUT	Filter Name	Specifies that the filter <Name> should be used as output filter
FILTER	Filter Name	Specifies that the filter <Name> should be used both as input and output filter

Chapter 3. Functions

This chapter describes how functions are linked to the logic tier. It introduces a language to write database transaction functions and a language to describe atomic data type normalizations and gives an overview on the language bindings available for Wolframe.

After reading this chapter you should be able to write functions of the Wolframe logic tier on your own.

Be aware that you have to configure a programming language of the logic tier in Wolframe before using it. Each chapter introducing a programming language will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

3.1. Functions in Transaction Definition Language (TDL)

3.1.1. Introduction

For the description of transactions Wolframe provides the transaction definition language (TDL) introduced here. Wolframe transactions in TDL are defined as functions in a transactional context. This means that whatever is executed in a transaction function belongs by default to a database transaction with an automatic commit on function completion if not explicitly defined otherwise by the caller. Errors lead to an automatic abort of the database transaction.

A TDL transaction function takes a structure as input and returns a structure as output. The Wolframe database interface defines a transaction as context where the input is passed as a structure and the output is fetched from it as a structure.

TDL is a language to describe the building of transaction input and the building of the result structure from the database output. It defines a transaction as a sequence of instructions on multiple data. An instruction is either described as a single embedded database command in the language of the underlying database, a name of a function declared in the database (e.g. a PLSQL function) or a TDL subroutine call working on multiple data.

Working on multiple data means that the instruction is executed for every item of an input set. This set can consist of the set of results of a previous instruction or a selection of the input of the transaction function. A "for each" selector defines the input set as part of the instruction.

Each instruction result can be declared as being part of the transaction result structure. The language has no control structures and is therefore not a general purpose programming language. It just offers some mapping of the input to commands and from the command results back to the output.

To convert input data the transaction definition language defines a preprocessing section where globally defined Wolframe functions can be called for the selected input. To build an output structure that cannot be modelled with a language without control structures and recursion, TDL provides the possibility to call a globally defined function as filter for postprocessing of the result of the transaction function.

The TDL is case insensitive. For clearness and better readability TDL keywords are written in uppercase here.

3.1.2. Examples

First Example

Our first example selects a key from a table and inserts it into a list. The substitute "$1" refers to the first column of the query result "SELECT id FROM UserTable ..". The substitute "$(name)" in the first statement refers to the toplevel element 'name' of the input. In the second statement it refers to the "name" element of the "item" selection in the query, e.g. for each element with the path "/item/name" (name being unique per item and item describing a set of items).

```
TRANSACTION insertItem
BEGIN
    INTO user
        DO UNIQUE NONEMPTY SELECT id FROM UserTable
            WHERE name = $(name);
    FOREACH item
        DO INSERT INTO ItemTable (ownerid,name)
            VALUES ($RESULT.id,$(name));
END
```

Preprocessing Input

This example selects identifiers 'id' of all users matched by their normalized name. The normalization of the name is done in a preprocessing step.

```
TRANSACTION selectPersonId
PREPROCESS BEGIN
    FOREACH person INTO normalizedName
        DO normalize_diachars( name );
END
BEGIN
    FOREACH person INTO user
        DO UNIQUE NONEMPTY
            SELECT id FROM UserTable
                WHERE name = $(normalizedName);
END
```

Using Variables

This example returns the 'id' of a country matched by their normalized name. The normalization of country names is done in a preprocessing step. The result is printed into a structure 'country/code'.

```
TRANSACTION selectCountry
PREPROCESS BEGIN
    INTO country/normalizedName DO normalize_diachars( country/name );
END
BEGIN
    DO UNIQUE NONEMPTY SELECT id FROM CountryTable
        WHERE name = $(country/normalizedName);
    KEEP AS COUNTRY;
        -- assigning the result with column name
        --   'id' to the variable 'countrycode'
    RESULT INTO country
        -- building a result block with name 'country'
    BEGIN
```

```
        INTO code PRINT $COUNTRY.id;
        -- printing the variable 'countrycode' with tag 'code'
        --   into the 'country' result block
    END
END
```

Subroutines

This example calls a subroutine with one argument to map the output.

```
SUBROUTINE selectFamilyMembers( id)
BEGIN
    INTO member
        DO SELECT surname,prename FROM PersonTable
            WHERE id = $PARAM.id;
END

TRANSACTION getFamilyMembers
BEGIN
    FOREACH person INTO . DO selectFamilyMembers( id);
END
```

3.1.3. Language Description

A TDL program consists of subroutine declarations and exported transaction function declarations. Subroutines have the same structure as transaction function blocks but without pre- and postprocessing and authorization method declarations.

Subroutines

A subroutine declaration starts with the Keyword `SUBROUTINE` followed by the subroutine name. After this header and an optional result substructure name declaration, the code definition block follows starting with `BEGIN`, ending with `END`.

```
SUBROUTINE <name>
RESULT INTO <result-block-name>
BEGIN
 ...<instructions>...
END
```

The line with the RESULT INTO declaration is optional.

Transaction Function Declarations

A transaction function declaration starts with the Keyword `TRANSACTION` followed by the name of the transaction function. This name identifies the function globally. The body of the function contains the following parts:

- `AUTHORIZE ( <auth-function>, <auth-resource> )`

This optional definition is dealing with authorization and access rights.

- RESULT INTO <result-block-name> FILTER <post-filter-name>

  This optional declaration defines a top level tag for the result structure and optionally a post processing step as filter referenced by name. It is similar as the construct in the subroutine description except that one cannot specify a postprocessing filter in a subroutine.

- PREPROCESS BEGIN <...preprocessing instructions...> END

  This optional block contains instructions on the transaction function input. The result of these preprocessing instructions are put into the input structure, so that they can be referred to in the main code definition block of the transaction. We can call any global normalization or form function in the preprocessing block to enrich or transform the input to process.

- BEGIN <...instructions...> END

  The main processing block starts with BEGIN and ends with END. It contains all the database instructions needed for completing this transaction.

The following pseudo code snippet shows the explained building blocks together:

```
TRANSACTION <name>
AUTHORIZE ( <auth-function>, <auth-resource> )
RESULT INTO <result-block-name> FILTER <post-filter-name>
PREPROCESS BEGIN
 ...<preprocessing instructions>...
END
BEGIN
 ...<instructions>...
END
```

The lines with AUTHORIZE and RESULT INTO and FILTER declarations are optional. So is the preprocessing block PREPROCESS..BEGIN..END. A simpler transaction function looks like the following:

```
TRANSACTION <name>
BEGIN
 ...<instructions>...
END
```

Main Processing Instructions

Main processing instructions defined in the main execution block of a subroutine or transaction function consist of three parts in the following order terminated by a semicolon ';' (the order of the INTO and FOREACH expression can be switched):

- INTO <result substructure name>

  This optional directive specifies if and where the results of the database commands should be put into as part of the function output. In subroutines this substructure is relative to the current substructure addressed in the callers context. For example a subroutine with an "INTO myres" directive called by a subroutine with an "INTO output" directive will write its result into a substructure with path "output/myres".

- `FOREACH <selector>`

  This optional directive defines the set of elements on which the instruction is executed one by one. Specifying a set of two elements will cause the function to be called twice. An empty set as selection will cause the instruction to be ignored. Without quantifier the database command or subroutine call of the instruction will be always be executed once.

  The argument of the FOREACH expression is either a reference to the result of a previous instruction or a path selecting a set of input elements.

  Results of previous instructions are referenced either with the keyword RESULT referring to the result set of the previous command or with a variable naming a result set declared with this name before.
  Input elements are selected by path relative to the path currently selected, starting from the input root element when entering a transaction function. The current path selected and the base element of any relative path calculated in this scope changes when a subroutine is called in a FOREACH selection context. For example calling a subroutine in a 'FOREACH person' context will cause relative paths in this subroutine to be sub elements of 'person'.

- `DO <command>`

  Commands in an instruction are either embedded database commands, named database functions (e.g. PLSQL functions) or subroutine calls. Command arguments are either constants or relative paths from the selector path in the FOREACH selection or referring to elements in the result of a previous command. If an argument is a relative path from the selector context, its reference has to be unique in the context of the element selected by the selector. If an argument references a previous command result it must either be unique or dependent an the FOREACH argument. Results that are sets with more than one element can only be referenced if they are bound to the FOREACH quantifier.

Main Processing Example

The following example illustrate how the FOREACH,INTO,DO expressions in the main processing block work together:

```
TRANSACTION insertCustomerAddresses
BEGIN
    DO SELECT id FROM Customer
        WHERE name = $(customer/name);
    FOREACH /customer/address
        DO INSERT INTO Address (id,address)
        VALUES ($RESULT.id, $(address));
END
```

Preprocessing Instructions

Preprocessing instructions defined in the PREPROCESS execution block of a transaction function consist similar to the instructions in the main execution block of three parts in the following order terminated by a semicolon ';' (the order of the INTO and FOREACH expression can be switched):

- `INTO <result substructure name>`

  This optional directive specifies if and where the results of the preprocessing commands should be put into as part of the input to be processed by the main processing instructions. The relative paths of the destination structure are calculated relative to a FOREACH selection element.

- FOREACH <selector>

  This optional directive defines the set of elements on which the instruction is executed one by one. The preprocessing command is executed once for each element in the selected set and it will not be executed at all if the selected set is empty.

- DO <command>

  Commands in an instruction are function calls to globally defined form functions or normalization functions. Command arguments are constants or relative paths from the selector path in the FOREACH selection. They are uniquely referencing elements in the context of a selected element.

Preprocessing Example

The following example illustrate how the "FOREACH, INTO, DO" expressions in the main processing block work together:

```
TRANSACTION insertPersonTerms
PREPROCESS BEGIN
    FOREACH //address/* INTO normalized
        DO normalizeStructureElements(.);
    FOREACH //id INTO normalized
        DO normalizeNumber(.);
END
BEGIN
    DO UNIQUE SELECT id FROM Person
        WHERE name = $(person/name);
    FOREACH //normalized DO
        INSERT INTO SearchTerm (id, value)
        VALUES ($RESULT.id, $(.));
END
```

Selector Path

An element of the input or a set of input elements can be selected by a path. A path is a sequence of one of the following elements separated by slashes:

- Identifier
  An identifier uniquely selects a sub element of the current position in the tree.

- `*`

  An asterisk selects any sub element of the current position in the tree.

- `..`

  Two dots in a row select the parent element of the current position in the tree.

- `.`

  One dots selects the current element in the tree. This operator can also be useful as part of a path to force the expression to be interpreted as path if it could also be interpreted as a keyword of the TDL language (for example `./RESULT`).

A slash at the beginning of a path selects the root element of the transaction function input tree. Two subsequent slashes express that the following node is (transitively) any descendant of the current node in the tree.

Paths can appear as argument of a FOREACH selector where they specify the set of elements on which the attached command is executed on. Or they can appear as reference to an argument in a command expression where they specify uniquely one element that is passed as argument to the command when it is executed.

When used in embedded database statements, selector paths are referenced with $(<path expression>). When used as database function or subroutine call arguments path expressions can be used in plain without '$' and '(' ')' markers. These markers are just used to identify substitution entities.

Path Expression Examples

The following list shows different ways of addressing an element by path:

- `/`

  Root element

- `/organization`

  Root element with name "organization"

- `/organization/address/city`

  Element "city" of root "organization" descendant "address"

- `//id`

  Any descendant element with name "id" of the current element

- `//person/id`

  Child with name "id" of any descendant "person" of the current element

- `/.//id`

  Any descendant element with name "id" of the root element

- `/address/*`

  Any direct descendant of the root element "address"

- `.`

  Currently selected element

Path Usage Example

This example shows the usage of path expression in the preprocessing and the main processing part of a transaction function:

```
TRANSACTION selectPerson
PREPROCESS BEGIN
    FOREACH /person/name INTO normalized DO normalizeName( . );
    FOREACH /person INTO citycode DO getCityCode( city );
END
BEGIN
    FOREACH person
        DO INSERT INTO Person (Name,NormalizedName,CityCode)
            VALUES ($(name),$(name/normalized),$(citycode));
```

```
END
```

Referencing Database Results

Database results of the previous instruction are referenced with a '$RESULT.' followed by the column identifier or column number. Column numbers start always from 1, independent from the database ! So be aware that even if the database counts column from 0 you have to use 1 for the first column.

As already explained before, database result sets of cardinality bigger than one cannot be addressed if not bound to a FOREACH selection. In statements potentially addressing more than one result element you have to add a `FOREACH RESULT` quantifier.

For addressing results of instructions preceding the previous instruction, you have to name them (see next section). The name of the result can then be used as FOREACH argument to select the elements of a set to be used as base for the command arguments of the instruction. Without binding instruction commands with a FOREACH quantifier the named results of an instruction can be referenced as `$<name>.<columnref>`. For example as `$person.id` for the column with name 'id' of the result named as 'person'.

The 'RESULT.' prefix in references to the previous instruction result is a default and can be omitted in instructions that are not explicitly bound to any other result than the last one. So the following two instructions are equivalent:

```
DO SELECT name FROM Company WHERE id = $RESULT.id
DO SELECT name FROM Company WHERE id = $id
```

and so are the following two instructions:

```
FOREACH RESULT DO SELECT name FROM Company WHERE id = $RESULT.id
FOREACH RESULT DO SELECT name FROM Company WHERE id = $id
```

The result name prefix of any named result can also be omitted if the instruction is bound to a FOREACH selector naming the result. So the following two statements in the context of an existing database result named "ATTRIBUTES" are equivalent:

```
FOREACH ATTRIBUTES DO SELECT name FROM Company WHERE id = $ATTRIBUTES.id
FOREACH ATTRIBUTES DO SELECT name FROM Company WHERE id = $id
```

Naming Database Results

Database results can be hold and made referenceable by name with the declaration `KEEP AS <resultname>` following immediately the instruction with the result to be referenced. The identifier <resultname> references the result in a variable reference or a FOREACH selector expression.

Named Result Example

This example illustrates how a result is declared by name and referenced:

```
TRANSACTION selectDevices
BEGIN
```

```
    SELECT id FROM DevIdMap WHERE name = $(device/name);
    KEEP AS dev;
    FOREACH dev SELECT key,name,registration FROM Devices WHERE sid=$id;
END
```

Referencing Subroutine Parameters

Subroutine Parameters are addressed like results but with the prefix `PARAM.` instead of `RESULT.` or a named result prefix. "PARAM." is reserved for parameters. The first instruction without FOREACH quantifier can reference the parameters without prefix by name.

```
SUBROUTINE selectDevice( id)
BEGIN
 INTO device
  DO SELECT name FROM DevIdMap
   WHERE id = $PARAM.id;
END

TRANSACTION selectDevices
BEGIN
 DO selectDevice( id );
END
```

Constraints on database results

Database commands returning results can have constraints to catch certain errors that would not be recognized at all or too late otherwise. For example a transaction having a result of a previous command as argument would not be executed if the result of the previous command is empty. Nevertheless the overall transaction would succeed because no database error occurring during execution of the commands defined for the transaction.

Constraints on database results are expressed as keywords following the DO keyword of an instruction in the main processing section.

The following list explains the result constraints available:

- `NONEMPTY`

  Declares that the database result for each element of the input must not be empty.

- `UNIQUE`

  Declares that the database result for each element of the input must be unique, if it exists. Result sets with more than one element are refused but empty sets are accepted. If you want to declare each result to have to exist, you have to put the double constraint "UNIQUE NONEMPTY" or "NONEMPTY UNIQUE".

Example with Result Constraints

This example illustrates how to add result constraint for database commands returning results:

```
TRANSACTION selectCustomerAddress
```

```
BEGIN
    DO NONEMPTY UNIQUE SELECT id FROM Customer
        WHERE name = $(customer/name);
    INTO address DO NONEMPTY SELECT street,city,country
        FROM Address WHERE id = $id;
END
```

Rewriting Error Messages for the Client

Sometimes internal error messages are confusing and are not helpful to the user that does not have a deeper knowledge about the database internals. For a set of error types it is possible to add a message to be shown to the user if an error of a certain class happens. The instruction `ON ERROR <errorclass> HINT <message>;` following a database instruction catches the errors of class <errorclass> and add the string <message> to the error message show to the user.

We can have many subsequent ON ERROR definitions in a row if the error classes to be caught are various.

Database Error HINT Example

The following example shows the usage HINTs in error cases. It catches errors that are constraint violations (error class CONSTRAINT) and extends the error message with a hint that will be shown to the client as error message:

```
TRANSACTION insertCustomer
BEGIN
    DO INSERT INTO Customer (name) VALUES ($(name));
    ON ERROR CONSTRAINT HINT ". Customers must have a unique name.";
END
```

On the client side the following message will be shown:

```
unique constaint validation in transaction 'insertCustomer'. Customers must hav
```

Substructures in the Result

We already learned how to define substructures of the transaction function result with the `RESULT INTO` directive of an OPERATION or TRANSACTION. But `RESULT INTO` blocks can also be defined locally as sub blocks in the main processing block. A sub-block in the result is declared with

```
RESULT INTO <resulttag>
BEGIN
 ...<instruction list>...
END
```

All the results of the instruction list that get into the final result will be attached to the substructure with name <resulttag>. The nesting of result blocks can be arbitrary and the path of the elements in the result follows the scope of the sub-blocks.

Explicit Definition of Elements in the Result

The result of a transaction consists normally of database command results that are mapped into the result with the attached INTO directive. For printing variable values or constant values you can in certain SQL databases use a select constant statement without specifying a table. Unfortunately select of constants might not be supported in your database of choice. Besides that explicit printing seems to be much more readable. The statement `INTO <resulttag> PRINT <value>;` prints a value that can be a constant,variable or an input or result reference into the substructure named <resulttag>. The following artificial example illustrates this.

```
TRANSACTION doPrintX
RESULT INTO person
BEGIN
    INTO name PRINT 'jussi';
    INTO id PRINT '1';
END
```

3.2. Functions in .NET

3.2.1. Introduction

You can write functions for the logic tier of Wolframe in languages based on .NET (http://www.microsoft.com/net) like for example C# and VB.NET. Because .NET based libraries can only be called by Wolframe only as a compiled and not as an interpreted language, you have to build a module out of your function implementation.

The implementation of .NET calls is not yet available. The implementation exists as prove of concept in a test program. But Wolframe will provide .NET functions soon.

3.3. Functions in Python

3.3.1. Introduction

You can write functions for the logic tier of Wolframe in the Python programming language (http://www.python.org).

The implementation of Python calls is not yet available. But Wolframe will provide Python functions soon.

3.4. Functions in Lua

3.4.1. Introduction

You can write functions for the logic tier of Wolframe with Lua. Lua is a scripting language designed, implemented, and maintained at PUC-Rio in Brazil by Roberto Ierusalimschy, Waldemar Celes and Luiz Henrique de Figueiredo (see http://www.lua.org/authors.html). A description of Lua is not provided here. For an introduction into programming with Lua see http://www.lua.org. The official manual which is also available as book is very good. Wolframe introduces some Lua interfaces to access input and output and to execute functions. The Lua application layer does not have to deal with protocol or encryption issues.

3.4.2. Declaring Functions

For Lua you do not have to declare anything in addition to the Lua script. If you configure a Lua script as program, all global functions declared in this script are declared as form functions in Wolframe. For avoiding name conflicts you should declare private functions of the script as `local`.

3.4.3. Wolframe Provider Library

Wolframe lets you access objects of the global context through a library called `provider` offering the following functions:

Name	Parameter	Returns
form	name of the form	an instance of a form
normalizer	name of the normalize function	normalize function defined in a Wolframe normalizer program
formfunction	name of the function	form function defined in a Wolframe program or module

3.4.4. Wolframe Global Objects

Besides the provider library Wolframe defines the following objects global in the script execution context:

Name	Description
transaction	transaction context if not relying on the auto-commit of a transaction function
scope	function with an iterator as argument returning an iterator for iterating on the currently visited node and its children taking ownership on this sub iterator scope
logger	object with methods for logging or debugging

3.4.5. Using Forms

The provider function `provider.form( )` with the name of the form as string as parameter returns an empty instance of a form. It takes the name of the form as string argument. If you for example have a form configured called "employee" and you want to create an employee object from a Lua table, you call

```
bcf = provider.form( "employee" )
bcf:fill( {surname='Hans', name='Muster', company='Wolframe'} )
```

The first line creates the data form object. The second line fills the data into the data form object.

The form method `fill` takes a second optional parameter. Passing "strict" as second parameter enforces a strict validation of the input against the form, meaning that attributes are checked to be attributes (when using XML serialization) and non optional elements are checked to be initialized. Passing "complete" as second parameter forces non optional elements to be checked for initialization but does not distinguish between attributes and content values. "relaxed" is the default and checks only the existence of filled-in values in the form.

Given the following validation form:

```
FORM Employee
{
   employee
   {
       ID !@int                    ; Internal customer id (mandatory)
       name !string                ; Name of the customer (mandatory)
       company string              ; Company he is working for (optional)
   }
}
```

the call of `fill` in the following piece of code will raise an error because some elements of the form ('ID' and 'name') are missing in the input:

```
bc = provider.form( "employee" ):fill( {company='Wolframe'}, "strict" )
```

To access the data in a form there are two form methods available. `get()` returns an iterator on the form data similar to the iterator on input. Same as for input there is a method `value()` that returns the form data as Lua data structure (a Lua table or atomic value).

3.4.6. Form Functions

For calling transactions or built-in functions loaded as modules the Lua layer defines the concept of functions. The provider function `provider.formfunction` with the name of the function as argument returns a Lua function. This function takes a table or an iterator as argument and returns a data form structure. The data in the returned form data structure can be accessed with `get()` that returns an iterator on the content and `value()` that returns a Lua table or atomic value.

If you for example have a transaction called "insertEmployee" defined in a transaction description program file declared in the configuration called "insertEmployee" and you want to call it with the 'employee' object defined above as input, you do

```
f = provider.formfunction( "insertEmployee")
res = f ( {surname='Hans', name='Muster', company='Wolframe'} )
t = res:value()
output:print( t[ "id" ] )
```

The first line creates the function called "insertEmployee" as Lua function. The second calls the transaction, the third creates a Lua table out of the result and the fourth selects and prints the "id" element in the table.

3.4.7. List of Lua Objects

This is a list of all objects and functions declared by Wolframe:

Table 3.1. Data forms declared by DDL

Method Name	Arguments	Returns	Description
get		iterator (*)	Returns an iterator on the form elements

Method Name	Arguments	Returns	Description
value		Lua table	Returns the contents of the data form as Lua table or atomic value
__tostring		string	String representation of form for debugging
name		string	Returns the global name of the form.
fill	Lua table or iterator (*), optional validation mode (**)	the filled form (for concatenation)	Validates input and fills the input data into the form.

(*) With iterator we mean an iterator as introduced in previous sections, returned by a Wolframe Lua object 'get()' method.

(**) "strict" (full validation), "complete" (only check for all non optional elements initialization) or "relaxed" (no validation except matching of input to elements)

Table 3.2. Data forms returned by functions

Method Name	Returns	Description
get	iterator (*)	Returns an iterator on the form elements
value	Lua table or atomic value	Returns the contents of the data form as Lua table or atomic value
__tostring	string	String representation of form for debugging

(*) With iterator we mean an iterator as introduced in previous sections, returned by a Wolframe Lua object 'get()' method.

Table 3.3. Global Transaction Context

Method Name	Description
transaction.begin	Starts a transaction
transaction.commit	Commit of a transaction
transaction.rollback	Rollback of a transaction

Table 3.4. Logger functions

Method Name	Arguments	Description
logger.printc	arbitrary list of arguments	Print arguments to standard console output
logger.print	loglevel (string) plus arbitrary list of arguments	log argument list with defined log level

Table 3.5. Global functions

Function Name	Arguments	Description
provider.form	name of form (string)	Returns an empty data form object of the given type

Function Name	Arguments	Description
provider.formfunction	name of function (string)	Returns a lua function to execute the Wolframe function specified by name
provider.normalizer	type name	Returns a Lua function for normalizing values of the type specified (as used in forms)
scope	iterator (*)	Returns an iterator to iterate till the end of the current tag (**)

(*) With iterator we mean an iterator as introduced in previous sections, returned by a Wolframe Lua object 'get()' method.

(**) The iterator of a defined scope must be consumed completely before consuming anything of the parent iterator. Otherwise it may lead to unexpected results because they share some part of the iterator state.

3.5. DDL Data Types

3.5.1. Introduction

Forms defined by DDLs are typed. The only predefined type is 'string' that is neither validated nor transformed for processing in any way. The types that can be used in DDLs are defined in files with the extension ".wnmp" and defined as programs in the configuration.

A .wnmp file contains assignments of a type name to sequences of normalization function calls where the first takes the initial input. The output of a function in the sequence gets the input of the next one and the final output for the last one. Each normalization step validates the input as atomic type (arithmetic,string,etc.) and transforms it to another atomic type.

3.5.2. Example

The example defines 3 numeric types including trimming of the input string for mode tolerant parsing and a string type that is converted to lowercase as normalization.

```
int=string:trim,number:integer;
uint=string:trim,number:unsigned;
float=string:trim,number:float;
name=string:trim,lcname;
```

3.5.3. Language Description

Each type declaration in a `.wnmp` file starts with an identifier followed by a an assignment operator '='. The left side identifier specifies the name of the type. This type name can be used in a DDL as name instead of the built in type `string`. When doing so the referenced token is validated and normalized with the comma separated sequence of function references on the right side of the assignment before being assigned as value to the element in the form declared to be of this type.

The function names are either identifiers or have two parts separated by ':'. The left part is the name of the module declaring the function. The right part is the name of the function in the module with the name in front of the ':'. If the function is declared as one single identifier then the module referred to is the previous one in the sequence before the identifier. The 4th line in the example declaration list above for example refers to the 'lcname' function in the 'string' normalization module.

Functions can have a comma separated list of constant arguments in brackets ('(' and ')') depending on the function type. An integer type for example could have the maximum number of digits of the integer type.

3.5.4. Configuration

For declaring and using a the .wnmp file of our example above, you have to load the module 'mod_normalize_string' and the module 'mod_normalize_number'. For this you add the following two lines to the LoadModules section of your Wolframe configuration:

```
module mod_normalize_number
module mod_normalize_string
```

You also have to add the declaration of the program "example.wnmp" (listing example above) to the Processor section of the configuration.

```
program example.wnmp
```

3.6. Functions in C++

3.6.1. Introduction

You can write functions for the logic tier of Wolframe with C++. Because C++ is supported by Wolframe only as a compiled and not as an interpreted language, you have to build a module out of your function implementation.

3.6.2. Declaring Functions

...

3.6.3. Input/Output Data Structures

For defining input and output parameter structures in C++ you have to define the structure and its serialization description. The serialization description is a static function `getStructDescription` without arguments returning a const structure that describes what element names to bind to which structure elements.

The following example shows a form function parameter structure defined in C++.

Header file "customers.hpp": Declare the structure and the serialization description of the structure. Structures may contain structures with their own serialization description.

```
#include "serialize/struct/filtermapBase.hpp"
#include <string>

namespace _Wolframe {
namespace example {

struct Customer
{
       int ID;                         // Internal customer id
       std::string name;               // Name of the customer
       std::string canonical_Name;     // Customer name in canonical form
       std::string country;            // Country
       std::string locality;           // Locality

       static const serialize::StructDescriptionBase* getStructDescription();
};

}}//namespace
```

Source file "customers.cpp". Declare 'ID' as attribute and name, canonical_Name, country, locality as tags. The '--' operator marks the end of attributes section and the start of content section.

```
#include "serialize/struct/filtermapDescription.hpp"

using namespace _Wolframe;

namespace {
struct CustomerDescription :public serialize::StructDescription<Customer>
{
 CustomerDescription()
 {
  //
  (*this)
  ("ID", &Customer::ID)
  --
  ("name", &Customer::name)
  ("canonical_Name", &Customer::canonical_Name)
  ("country", &Customer::country)
  ("locality", &Customer::locality)
  ;
 }
};

const serialize::StructDescriptionBase* Customer::getStructDescription()
{
 static CustomerDescription rt;
 return &rt;
}
```

Chapter 4. Forms

Forms are data structures used to validate input and output data and to do some basic normalizations in order to make data accessible in a uniform way. Forms are defined in a data definition language (DDL) and translated by a compiler at startup. Those compilers are defined as loadable modules.

This chapter describes how form data schemas are linked to the logic tier. It introduces a data description language (DDL) called `simpleform` that allows you to specify data schemas with the validation and normalization of atomic types. It also describes the Wolframe module concept for form descriptions that allows you to add a compiler for your existing data schemas.

After reading this chapter you should be able to write data forms of Wolframe of the logic tier in the `simpleform` data description language on your own. You should also know how a new data description language (DDL) could be added.

Be aware that you have to configure a data description language type (DDL compiler) of the logic tier in Wolframe before using it. Each chapter introducing a data form description language will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

4.1. Form Data Definition Languages

4.1.1. Introduction

Form data structures can be defined in a DDL (Data Definition Language). It depends very much on the application what DDL is best to use. Users may already have their data definitions defined in a certain way. The form DDL can be defined in the way you want. Wolframe offers a plugin mechanism for DDL compilers and provides examples of such compilers. You configure the DDL sources to load and the compiler to use.

With the DDL form description we get a deserialization of some content into a structure and a serialization for the output. We get also a validation and normalization procedure of the content by assigning types to atomic form elements that validate and normalize the data elements. Most of the business transactions should be doable as input form description, output form description and a transaction that maps input to output without control flow aware programming (direct map).

All types of data forms introduced here are equivalent in use for all programs.

4.1.2. Forms in Simpleform DDL

As example of a form DDL we provide the *simpleform* DDL. The format is based on the "INFO"-format introduced by Marcin Kalicinski for the boost property tree library. We used this library to show an example that is easy to understand and small enough. The format uses key value pairs separated with spaces for atomic elements and curly brackets '{' '}' to describe structures. The key represents the name of the element and the value represents the type of the element. The type is defined by a typename and some operators that describe additional properties.

Each form declaration starts with a keyword 'FORM' or 'STRUCT'. The difference between 'FORM' and 'STRUCT' is that the later is used for declarations that are only referenced inside the same file as sub structure reference, while 'FORM' declares a structure to be exported as global form declaration. This header is followed by the structure declaration inside curly brackets '{' '}'.

There is only one predefined data type known in *simpleform* DDL: "string". All other data types are defined as sequence of 'normalizer' functions in a normalize definition file. The 'normalizer' functions assigned to a type validate the value and transform it to its normalized form.

The following element attributes are known in *simpleform* DDL:

Table 4.1. element attributes in simpleform

Attribute	Location	Description
@	prefix of data type	Expresses that the element is an attribute and not a content element of the structure. This has only influence on the XML or similar representation of the form content
?	prefix of data type	Expresses that the element is optional also in strict validation
^	prefix of form name	Expresses that the element is optional and refers to a structure defined in the same module that is expanded only if the element is present. With this construct it is possible to define recursive structures like trees.
!	prefix of data type	Expresses that the element is always mandatory (also in non strict validation)
[]	prefix of data type	Expresses that the element is an array of this type
[]	without data type	Expresses this element is an array of structures and that the structure defined describes the prototype (initialization) element of the array.
(..)	postfix of data type	Expresses that '..' (represents any valid value not containing brackets) is the default initialization value of this element.

Using a single underscore as typename ('_') means that element is embedded into the structure without being referenceable by name. In case of an atomic value it means that value represents the content value of the structure. In case of a substructure it means that the structure inherits the embedded elements of the substructure.

The following example shows a form defined in *simpleform* DDL.

```
FORM Customer
{
   customer
   {
       ID !@int                   ; Internal customer id (mandatory)
       name string                ; Name of the customer
       canonical_Name string      ; Customer name in canonical form
       country string             ; Country
       locality ?string           ; Locality (optional)
   }
}
```

Chapter 5. Filters

Filters describe the transformation of serialized data to a unified serialization of hierarchical structured data and back. The application does not care about data formats as long as there exists a filter providing the unified form of serialization.

This chapter describes how filters for different data formats are linked to the logic tier. For each data format supported by Wolframe one or more filter type is introduced.

After reading this chapter you should be able to handle different document formats and encodings in the logic tier of Wolframe. You will know how to add programs for scriptable filters like XSLT.

Be aware that you have to configure a data filter of the logic tier in Wolframe before using it. Each chapter introducing a filter type will have a section that describes how the server configuration of Wolframe has to be extended for its availability.

5.1. XML Filter

5.1.1. Introduction

You can use TDL for data filters in the logic tier of Wolframe. There are the following variants of TDL filters available:

- libxml2 (http://www.xmlsoft.org) or
- textwolf (http://www.textwolf.net)

5.1.2. Character Set Encodings

The libxml2 and the textwolf filter support at least the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or
- UTF-16LE or
- UTF-16 (UTF-16BE) or
- UTF-32LE (UCS-4LE) but only with textwolf or
- UTF-32 (UTF-32BE or UCS-4BE) or
- ISO 8859 (code pages '1' to '9')

5.1.3. Configuration

For using an XML filter based libxml2, you have to load the module 'mod_filter_libxml2'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_libxml2
```

For using an XML filter based textwolf, you have to load the module 'mod_filter_textwolf'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_textwolf
```

5.2. JSON Filter

5.2.1. Introduction

You can use JSON for data filters in the logic tier of Wolframe. The standard JSON filter of Wolframe is called cjson and based on the library cJSON (http://sourceforge.net/projects/cjson) from Dave Gamble.

5.2.2. Character Set Encodings

Without explicitly specified, the cjson filter support the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or
- UTF-16LE or
- UTF-16 (UTF-16BE) or
- UTF-32LE (UCS-4LE) or
- UTF-32 (UTF-32BE or UCS-4BE) or

5.2.3. Configuration

For using the JSON filter based cJSON, you have to load the module 'mod_filter_cjson'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_cjson
```

5.3. XSLT Filter

5.3.1. Introduction

You can use XSLT for data filters in the logic tier of Wolframe. The XSLT filter of Wolframe for is based on libxml2 (http://www.xmlsoft.org).

5.3.2. Character Set Encodings

Without explicitly specified, the XSLT filter support the following character set encodings. For character set encodings that are not in the list, please ask the Wolframe team.

- UTF-8 or
- UTF-16LE or

- UTF-16 (UTF-16BE) or
- UTF-32LE (UCS-4LE) or
- UTF-32 (UTF-32BE or UCS-4BE) or

5.3.3. Configuration

For using an XSLT filter based libxml2, you have to load the module 'mod_filter_libxml2'. For this you add the following line to the LoadModules section of your Wolframe configuration:

```
module mod_filter_libxml2
```

You also have to add the program of the XSLT filter into the Processor section of the configuration. The name of the filter is the filename of the XSLT filter program without path and extension. In our example the filter would be named invoice_ISOxxxx:

```
program invoice_ISOxxxx.xslt
```

Chapter 6. Testing and Defect Handling

In this chapter we learn how parts of a Wolframe application can be verified to work correctly. The basis for testing and debugging a Wolframe application is the command line tool wolfilter.

6.1. Using wolfilter

The command line program wolfilter allows you to call any Wolframe function or filter or mapping into a form structure on command line.

There are two possibilities to declare the items involved in the test. Either you pass the configuration with the option '--config' and the name of the command to execute or you declare the items one by one with program options. These two approaches are not mixable. Either you use '--config' or pass the parameters one by one. A try to mix both of them in one call is refused by wolfilter.

The following examples assume the input file name to be in.xml or in.json and the output file to be named out.xml or out.json respectively.

6.1.1. Testing a Filter

The following example shows the mapping through a libxml2 filter. Filters are tested by passing a dash '-' command to execute.

```
cat in.xml | wolfilter -f libxml2 -m mod_filter_libxml2 - > out.xml
```

The following example shows the processing of the input through an xslt filter and mapping the output through a token filter that shows the tokenization of the input by the input filter.

```
cat in.xml | wolfilter -i myfilter -o token\
 -m mod_filter_libxml2\
 -m mod_filter_token\
 -p myfilter.xslt - > out.xml
```

6.1.2. Testing a Form

The following example shows the mapping through a form defined with simpleform DDL. Mapping through forms is tested by passing the name of the form as command to execute.

```
cat in.xml | wolfilter -f libxml2 \
 -m mod_filter_libxml2\
 -p myform.sfrm MyForm > out.xml
```

we assume here that the form to use is defined in myform.sfrm and called MyForm.

6.1.3. Testing a Function

The following example shows the execution of a function written in Lua. A JSON filter is used for input and output.

```
cat in.xml | wolfilter -f cjson \
 -m mod_filter_cjson -m mod_command_lua \
 -m mod_command_directmap
 -p myfunc.lua MyFunc > out.json
```

we assume here that the exported function to call defined in myfunc.lua is called MyFunc.

Glossary

This is the glossary for the Wolframe Service Development Kit. Although it covers most of the terms used in the Wolframe world, some terms might be skipped if they are rarely used in the SDK context. These terms are explained in the Application Building Manual

External glossary

Data Definition Language
: A domain specific language for describing data structures

Wolframe glossary

Connection Handler
: Interface for the networking to one client/server connection during its whole lifetime.

Command Handler
: Interface for delegating processing of client protocol commands in a hierarchical way. A command handler is created by a connection handler or another command handler. During command execution the input/output of the connection is entirely handled by the command handler. Command Handlers are used to build the communication protocol processing as hierarchical state machine.

Network Input
: Data passed from the networking framework to the connection handler through its channel for processing.

Network Output
: Data passed back through the channel to the networking framework from the connection handler as output of processing.

Program
: A set of named units of description of processing or data in a source file. The source file is loaded and interpreted at server startup. It exports some unique symbols to address single units in a program library. All units in this library are usable in any context of the logic tier.

Transaction
: A transaction is a call of a database defined in a transaction program. A transaction either fails completely or succeeds as whole. Auditing is seen as part of the transaction. Transactions have an object as input and return an object or an error as result. Authorization tags that are checked against the user privileges of the connection can be attached to transactions.

Processor program
: A program that is executed by a command handler. It works with data input, data output and transaction execution as the only interfaces to the system. Exceptions, signals and events are handled by the processor in the background without showing them to the program. A program in this context is sequential.

Lua
: Lua (www.lua.org) is a scripting language. It is used in Wolframe as one language for writing programs. It distinguishes itself by being lightweight. It has to be lightweight because the language context has to be recreated with every start of a clients command execution for security reasons.

Filter
: Filters are attached to network input and output to read and write input in a well defined format. Filters let you process input and print output in an iterative way. Filters are loaded by the system at startup and have a unique name.

Form
: A form is a hierarchical description of typed data. Forms are used to create objects from a serialization and to validate input. Forms are defined in programs written in a DDL (Data Definition Language) or as declared as part of a build-in function API.

Direct Map	A direct map is a program described by a form for the input, a form for the output, and a transaction function. The input of a direct map transaction function is the object deserialization of the input xml defined by the input form. The result of the transaction function is returned as xml serialization of the transaction result defined by the output form. It is called direct map because no scripting is involved in this case.
Channel	The flow for a single connection. Not all objects have channels (e.g. databases).
Group	A set of objects of the same type seen as one single object for the objects that use it.
Unit	An element of a group. A group is a set of units.
Provider	An entity providing objects of a kind. Some providers are factories, but not all of them (e.g. the database provider)
End Of Data (EoD)	Marks the end of the data to be processed by a processor. End of data is marked with CR LF dot ('.') CR LF or LF dot LF. For passing lines with a dot ('.') at the start of a content line, the client has to escape an LF dot in the content with LF dot dot. This escaping applies also to the result returned to the client. So client has to unescape LF dot sequences by replacing them by a LF.
End Of Message (EoM)	Marks the end of a network input message. End of message is handled by the processor without showing up to the processor program (yield execution).
Application Reference Path	The application server defines this file path where all relative paths defined in the configuration refer to.
Yield execution	The processor yields execution and gives control back to the connection handler, when it cannot continue execution. The connection handler has to resume with an action returned to the networking framework, that fetches the resources needed by the processor to continue. (see also [http://en.wikipedia.org/wiki/Coroutine Wikipedia yield in coroutine]). Lua is able to yield execution.

Index

Appendix A. Language Grammars

A.1. Grammars

.1. EBNF of the Transaction Definition Language

Tokens

[1] Identifier ::= (['A'-'Z'] | ['a'-'z'] | ['0'-'9'] | '_') +

[2] Number ::= (['0'-'9']) +

[3] String ::= (['\"]([^'\"])+['\"]+) | (['\"']([^'\"'])+['\"']+)

[4] Selection ::= (('/' | '//' |) (Identifier | '..') (Selection |))

[5] EmbeddedArg ::= (String | '$' (Number | (Identifier | '(' Selection ')' | '[' Identifier ']'))

[6] FuncArg ::= (String | Selection | EmbeddedArg)

[7] PreprcArg ::= (String | Selection)

Statements

[8] Destination ::= 'INTO' Identifier

[9] Foreach ::= 'FOREACH' Selection

[10] EmbeddedOp ::= SQLStatement

[11] FuncArgLst ::= (FuncArg ',' FuncArgLst | FuncArg)

[12] NamedOp ::= Identifier '(' (FuncArgLst |) ')'

[13] Action ::= 'DO' (EmbeddedOp | NamedOp)

[14] Statement ::= (Destination |) (Foreach |) Action ';'

Pre-Processing Commands

[15] PreprcArgLst ::= (PreprcArg ',' PreprcArgLst | PreprcArg)

[16] PreprcOp ::= Identifier '(' (PreprcArgLst |) ')'

[17] PreprcCall ::= 'DO' (PreprcOp)

[18] PreprcStm ::= (Destination |) (Foreach |) PreprcCall ';'

Transactions

[19] PreprcStmLst ::= (PreprcStm | PreprcStm PreprcStmLst)

[20] PreprcBody ::= 'BEGIN' PreprcStmLst 'END'

[21] StatementLst ::= (Statement | Statement StatementLst)

[22] OpBody ::= 'BEGIN' StatementLst 'END'

[23] Operation ::= 'OPERATION' Identifier ('RESULT' 'INTO' Identifier |) OpBody

[24] Transaction ::= 'TRANSACTION' Identifier ['AUTHORIZE' '(' Identifier ',' Identifier ')'] ['PREPROCESS' PreprcBody] ['RESULT' 'INTO' Identifier] OpBody

[25] Program ::= (Transaction | Operation) (Program |)

.2. EBNF of the Normalizer Function Declarations

The declaration of a type starts with an identifier followed by an assignment operator '=' and a comma separated sequence of ':' separated identifier tuples, defining the name of the normalization module (1st) and the name of the function (2nd). The type declaration is closed with a semicolon ';'.

Tokens

[26] Identifier ::= (['A'-'Z'] | ['a'-'z'] | ['0'-'9'] | '_') +

Declarations

[27] Namespace ::= Identifier ':'
[28] Call ::= ((Namespace Identifier) | Identifier)
[29] CallSeq ::= ((Call ',' CallSeq) | Call)
[30] Typedef ::= Identifier '=' CallSeq ';'
[31] Program ::= ((Typedef Program) | Typedef)

Appendix B. GNU General Public License version 3

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.

c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
one line to give the program’s name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.