Mentor®
A Siemens Business

Mentor® LoLa® framework

User Guide

Development Drop 0.3.0 DRAFT INTERNAL USE ONLY
February 2018

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777 Telephone: 503.685.7000
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: http://supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

List of Examples

Chapter 1
LoLa introduction

LoLa is a Low Latency framework for autonomous driving

Brief

LoLa provides several layers and libraries to abstract Autonomous Driving Algorithms from HW and underlying operating systems.
Allows Algorithms to be developed platform independent.
It provides interface, component, and system description in a high level domain specific languages.

- It provides basic transport layers between components.
- It provides tracing and monitoring functions.
- It provides OS abstraction with generic runtime.

LoLa Components

- `LoLaCore`
- `mgc/core/lola` platform independent core functions see C++ interface
- `mgc/core/trace/` extra trace functionality
- `lolagen` LoLa code generator see Invoking the generator
- `Demo` example code using LoLa services

Delivered Artifacts (for LoLa users)

see LoLa Delivery Artifacts

Building LoLa FW from sources (Only for LoLa developers)

- clone this code repository: `cd ~; git clone ssh://git@stash.alm.mentorg.com:7999/adasfadac/fadac.git`
- create a build directory and build: `cd ~/lola && mkdir build && cd build && cmake ../ && make && sudo make install`

- demo is located in `~/lola/build/Demo/` folder. == LoLa Delivery Artifacts

LoLa framework released as debian package more detail see Degian Wiki: DebianPackage

`Development drop contains 2 Debian package:`

- Debug/LoLa_0.1.0.deb
- Release/LoLa_0.1.0.deb

```
The Debug LoLa package installs Debug x86 runtime Lola lib core.
The Release package installs Release x86 runtime Lola lib core.
```

Debian package content

`LoLa release consist of:`

- static libraries
- c and c++ header files
- demo source files
- LoLa code generator binary
- Cmake LoLa Package

Directory Tree

```
### bin
#   ### lolagen
### doc
#   ### LoLa.pdf
### include
#   ### mgc
#       ### core
#           ### lola
#           #   ### assert.h
#           #   ### binding.h
#           #   ### core.h
#           #   ### core.hpp
#           #   ### message_base.hpp
#           #   ### message_ptr.hpp
#           #   ### string.hpp
#           #   ### time.hpp
#           #   ### trace_binding.h
#           #   ### trace_dummy.h
#           #   ### trace.h
#           #   ### vector.hpp
#           ### trace
#               ### lola
#                   ### serialize_json.h
### lib
```

```
#   ### liblola_core.a
#   ### liblola_core-d.a
#   ### liblola_trace.a
#   ### liblola_trace-d.a
### share
    ### LoLa
    #   ### LoLaConfig.cmake
    #   ### LoLaConfigVersion.cmake
    #   ### LoLaGenerate.cmake
    #   ### LoLaTargets.cmake
    #   ### LoLaTargets-debug.cmake
    #   ### LoLaTargets-debug-home-inst.cmake
    ### src
        ### LoLa
            ### Demo1
                ### bsp
                #   ### bsp.h
                #   ### CMakeLists.txt
                #   ### posix
                #   #   ### bsp.c
                #   ### win32
                #       ### bsp.c
                ### c
                #   ### camera.c
                #   ### CMakeLists.txt
                #   ### console_handler.c
                #   ### lidar.c
                #   ### main.c
                #   ### perception.c
                #   ### radar.c
                #   ### sensor_fusion.c
                ### CMakeLists.txt
                ### cxx
                #   ### camera.cpp
                #   ### CMakeLists.txt
                #   ### console_handler.cpp
                #   ### lidar.cpp
                #   ### main.cpp
                #   ### mixin_keyboard_activated.h
                #   ### mixin_safe_printer.h
                #   ### perception.cpp
                #   ### radar.cpp
                #   ### sensor_fusion.cpp
                ### Interface
                    ### camera_component.fcdl
                    ### camera_message.ftdl
                    ### CMakeLists.txt
                    ### console_handler_component.fcdl
                    ### console_messages.ftdl
                    ### distance_reading.ftdl
                    ### hello_world.fsdl
                    ### lidar_component.fcdl
                    ### lidar_message.ftdl
                    ### perception_component.fcdl
```

```
### radar_component.fcdl
### radar_message.ftdl
### sensor_fusion_component.fcdl
### sensor_fusion_message.ftdl
```

On most Debian like linux distribution packages are installed in <install prefix> = /usr/local

- the <install prefix>/bin contains binaries LoLa installs : lolagen here this is the code generator for fsdl, ftdl, and fcdl
- the <install prefix>/doc LoLa installs documentation LoLa.pdf.
- the <install prefix>/include contains shared library headers LoLa installs under mgc/core all core runtime headers and under mgc/core/trace all extra tracing headers. these headers are automatically added to include path of the generated libs if LoLa cmake packages are used. User does not have specify includes directly.
- the <install prefix>/lib contains static shared runtime and trace LoLa libraries if delivered LoLa code generator functions used see <install prefix>/share/src/LoLa/Demo1/Interface/CMakeLists.txt than runtime libs are automatically linked againts the target executable where the interface lib is used. The user does not have to explicitly add it to the target link libraries in cmake.
- <install prefix>/share/LoLa contains the cmake Package for LoLa see more info about cmake config packages. User receives a fully functional LoLa cmake package which contains target requirements for LoLaCore lib and also gives cmake functions to generate interface libs. see : <install prefix>/share/LoLa/LoLaGenerate.cmake lola_gen_fxdl_cpp_interface and lola_gen_fxdl_c_interface for generating c++ and c11 interfaces respectively. The usage of the functions are demonstrated in <install prefix>/share/src/LoLa/Demo1/Interface/CMakeLists.txt. Since the LoLa cmake packeg installed in default cmake search paths users can use it by simply calling find_package(LoLa REQUIRED) in they high level cmake files.
- <install prefix>/share/src/LoLa/Demo1 contains a fully functional LoLa user code to demonstrate LoLa framework features in C and in C. `The Demo code can be compiled by calling in a tmp folder` *`cmake <install prefix>/share/src/LoLa/Demo1 && make`* `it will build in C folder the C HelloWord demo app and in cxx folder the C` HelloWord Demo app.

Chapter 2
Overview

The LoLa Generator is an esential part of the framework. It allows the user to express his intentions in a high level language and generates C/C++ code out of it. The generated code contains headers and source files. Headers contain signatures of the functions that the user has to implement and are ment to be included from the user's code. The generated source files should be compiled into a library and linked against the user code together with the LoLa core library.

The users describes the system using the high level language. This description contains the description of the types and components that make up the system. It also contains the specific instances of the components and the connection between these components. There are 3 high level languages to help the user describe the system.

Chapter 3
Invoking the generator

Usage

```
python lolagen.py [<global options>] <command> [<command options>]
```

<global options> Global options instruments the every commands:

-h --help	prints usage information
--version	prints the version number than exits
-v --verbose	makes the generator verbose
-q --quiet	makes the generator quite

<command> Commands can be one of the following:

genc	Generates C output
gencpp	Generates C++ output

<command options> The following options can be used

-h --help	prints usage information of the specified command
-i INPUT --input=INPUT	The input file for the generation. Can be an .fsdl, .fcdl or a .ftdl file
-o OUTPUT_DIR --output=OUTPUT_DIR	The directory where the output files will be created, if not provided it will use the current directory.

-a --all	If provided it generates the output for all imported files as well not just for the current input
-I IMPORT_PATH --import-path=IMPORT_PATH	Can be provided multiple times. When trying to load imported descriptor files the generator will also check these directories
-g --generate-typedesc	When generating the output for the .fsdl file it will also generate a file containing all type information for messages that can be used for logging purposes
-d --generate-dependencies	Instead of generating the regular output only generate a cmake file with two variables defined. One with all the inputs that will be used when called with the given parameters the other is all the other is all the output that will be generated.

Note

The generated C output will use and build on the C output of the generator. In order to use the generated C library the user must also generate, compile and link the generated C files as well.

Examples

```
python lolagen.py genc -all -g -i system.fsdl
```

This generates all C files that is required for the system defined by system.fsdl. It will also generate the system_typedesc.c which if linked with lola_core_debug lib, can be used to pretty print any message in the system.

```
python lolagen.py gencpp -all -i system.fsdl
```

This will generate all C files that is required for the system defined by system.fsdl. In order the use the C interface the C interface must also be generated

```
python lolagen.py genc -I Import/Folder -i message.ftdl -o Output/Folder
```

This will generate the message.h and message.c files only in the Output/Folder and will try to resolve imports using the .ftdl files found in the Import/Folder.

```
python lolagen.py genc -all -d -i system.fsdl -o dependencies.cmake
```

This will generate a dependencies.cmake file that will contain all input files (including imports) that would be used during the generation of system.fsdl and all output files that would be generated.

Chapter 4
Framework System Description Language (FSDL)

This language declares the instance of the components used by the system. It also describes the connections between these instances.

Example

```
import components           # can import component types form fcdl files
import other_components

system
{
    node NodeType : NodeName  # Nodetype is not used currently
    {
        config
        {
            cpu = 3     # number of thread to use (Experimental)
        }

        instance SourceComponent : Source1   # Instance of SourceComponent
        {
            connection
            {
                Output = SourceNetwork      # Which port is connected to which ne
            }
        }

        instance AveragerComponent : AvgInstance
        {
            config
            {
                number = 42                 # passed as parameters to the instanc
            }                               # The component has to define this as

            connection
            {
                Input = SourceNetwork
                Output = AvgNetwork
            }
        }

        instance SyncedLatestComponent : LatestInstance
        {
            connection
```

```
            {
                TimedMessages               # Synced ports have to specified toge
                {
                    config                  # this section is optional
                    {
                        tolerance = 0.01    # configuration option for synced por
                    }
                    Port1 = SourceNetwork   # have to connect each port individua
                    Port2 = SourceNetwork
                    Port3 = AvgNetwork
                }
            }
        }
    }
}
```

Structure

FSDL structure.

```
<import> <import file 1>
<import> <import file 2>
...
<import> <import file n>

system
{
    <node definition>
}
```

<import file>	Name of an FCDL file without the extension. Components defined in the files can be used to create instances in the system description.
<node definition>	Defines a node within the system. Currently only a single node can be defined.

Node

The system definition consists of nodes that describe a single processing chip in the system. A processing chip can have multiple cores and run multiple tasks parallel. A node description contains the instances created from the components and the connection between these instances.

Node description.

```
node <node type> : <node name>
{
    config
    {
        cpu = <number of parallel tasks>
    }
```

```
    <instance definition 1>
    <instance definition 2>
    ...
    <instance definition n>
}
```

<node type>	Type of the node that identifies the processing chip. It's not used currently, can be any valid id.
<node name>	Node name is a LoLaID.
<instance definition>	Defines an instance of a component in the system. See bellow.
<number of parallel tasks>	The number or parallel execution context that will be created for the node. This means that maximum this many components can run parallel during system execution.

Instances

Instances are made from the components defined in the imported FCDL files.

Instance description.

```
instance <component name> : <instance name>
{
    config
    {
        <parameter name> = <paramter value>
    }

    connection
    {
        <port name 1> = <network name>
        <port name 2> = <network name>
        ...
        <port name n> = <network name>
        <synced subscribe name>
        {
            config
            {
                <synced port configuration option> = <config value>
            }
            <synced subport 1> = <network name>
            ...
            <synced subport n> = <network name>
        }
    }
}
```

<component name>	Name of the component that will be instanciated

<instance name>	Name of the instance is a LoLaID
<parameter name>	Name of the parameter defined in the component description. Please note that the config section is optional.
<parameter value>	Value of the parameter. It has to be a valid C initialization that doesn't contain a newline. It's not checked by the generator. So the user might get error messages during the compilation of the generated C code if it's not valid.
<port name>	Name of the port defined in component description. Can be either a publish, a subscribe.
<network name>	Name of the network the port connectes to
<synced subscribe name>	Name of the synced subscribe
<synced port configuration option>	possible configuration option for a synced port. So far only option is: *tolerance* — sets the tolerance value of a timed synced port. The value is in milliseconds (float)
<synced subport>	Name of a port within a synced subscription

Components are connected via networks. The user don't have to define the network beforehand, they are automatically defined when referenced. If multiple ports connect to the same network the system will automatically connect all input ports to all of the output ports on that network. It means that if there is a message published on any of the output ports connected to that network, it will be received by all those subscribed to that network.

Chapter 5
Framework Component Description Language (FCDL)

This language lets the user describe the components he intends to use in the system. The user has to provide the implementations of these components. The file has the following basic strcuture:

Example

```
import messages       # should import used messages
import more_messages  # all imports in the imported files
                      # are automatically included as well

component Averager
{
    parameters
    {
        uint32_t : number = 2  # components can have paramters
    }

    interface
    {
        store Msg[3]           # during operation stores maximum of 3 Msg message
        publish Msg : Output   # send Msg on its port named Output
        subscribe Msg : Input  # receives Msg on its port named Input
    }

    lola
    {
        max_instance_count = 5
    }
}

component Source
{
    interface
    {
        publish Msg : Output  # component with a single output and no parameters
    }

    lola
    {
        max_instance_count = 3
    }
}
```

```
component SyncedLatest
{
    interface
    {
        subscribe timed : TimedMessages  # component with a sync policy and no ou
        {
            config
            {
                tolerance = 10.5 # tolerance value for the timed policy in millis
            }
            Msg1 : Port1[5]    # This is the primary channel with a queue length
            Msg2 : Port2[10]   # you'll get the messages that belong together bas
            Msg3 : Port3[10]
        }
    }
}
```

Structure

```
<import> <import file 1>
<import> <import file 2>
...
<import> <import file n>

<component definition 1>
<component definition 2>
...
<component definition n>
```

<import file>	Name of an FTDL file without the extension. Messages defined in the FTDL can be used in the component definitions.
<component definition>	Defines a component in the system.

Components

Components are the parts of the system that create and process messages. Components can have multiple instances so during the implementation avoid the usage of global variables. As different instances of components can potentially run parallel in separate threads at once, the usage of global shared data between instances should be avoided. Instead it's recommended to use the messaging system to share data between instances of components.

Component description.

```
component <component name>
{
    parameters
    {
        <parameter type> : <parameter name 1> [ = <default initialization>]
        <parameter type> : <parameter name 2> [ = <default initialization>]
```

```
        ...
        <parameter type> : <parameter name n> [ = <default initialization>]
    }

    interface
    {
        publish <message name> : <port name>
        .or.
        store <message name>[<count>]
        .or.
        subscribe <message name> : <port name>
        .or.
        subscribe <synced policy> : <synced port name>
        {
            config
            {
                <config name> = <config value>
            }
            <message name> : <port name 1>\[<queue length>\]
            <message name> : <port name 2>\[<queue length>\]
            ...
            <message name> : <port name n>\[<queue length>\]
        }
        ...
    }

    lola
    {
        <setting name> = <setting value>
    }
    ...
}
```

<component name>	Component name is a LoLaID
<parameter type>	Can be any of the basic types, array or vector. Can't be a `struct` or an `enum`.
<port name>	Port name is a LoLaID
<count>	Number of messages this component will store internally during runtime. This is required for calculating the number of preallocated messages in the framework.
<synced policy>	Synced policy decides the rules how the port will handle the incoming messages. It can be **master**, **timed**, **latest**
<synced port name>	Synced Port name is a LoLaID
<setting name>	name of a setting
<setting value>	value of a setting

Parameters can be defined for a component. These parameters will be passed to the user code during the construction of the component instances. Every instance can override the default value of the parameters.

The config inside the synced policy is optional. Currently only needed for the timed policy where you can specify the tolerance in milliseconds. The number can be a floating point number.

A component can only have a single subscribe statement in its interface definition, but with the usage of synced policies it still can receive data from multiple sources.

Components have a couple of settings that not as much describe the component as an interface, but rather tell the LoLa generator / framework some extra information about the component. Currently the only supported setting is "max_instance_count", which limits how many instance can be made from this component (in the FSDL). If not supplied, it is 1. == Framework Type Description Language (FTDL)

This language describes the types and the messages used by the components.

Example

```
import common_base  #can import other FTDL files

enum Numbers
{
    ZERO      # implicit value start with 0
    ONE  = 1  # explicit value
    TWO       # implicit values continue from the last value+1
    THREE
}

const uint32_t : MaxThings = 32  # constant

struct Simple
{
    bool : boolean       # bool value
    int16_t : shortNum   # 16 bit signed integer value
}

struct Basic
{
    int32_t : num        # basic signed integer
    uint32_t : unsNum    # basic unsigned integer
    float64_t : fltPoint # 64 bit floating point number
    string : str         # default fixed length string
    string[10] : str2    # explicitly defined fixed length string
    array string[Numbers.THREE] : arr[2]     # 2 length array of 3 length strings
    vector Numbers : numArray[Numbers.THREE] # max 3 length dynamic array of enum
}

struct Complex
```

```
{
    Basic : data                 # nesting structs is allowed
    vector Basic : dataArray[4]  # max 4 length dynamic array of structs
    int8_t : smallInt = 3        # default initialization
    array uint64_t : nums[2] = { 1, 2 } # default initialization of arrays
#
# vectors need to be initialized with first specifying the number of items in the
# and then listing the value of those members. Keep in mind that the number of it
# not exceed the capacity of the vector. The number of items in the initialiation
# should not exceed the specified size of the vector (excess items will be ignore
# must not exceed the capacity of the vector.
#
# vector stored_type : name[capacity] = { size, { items } }
#
    vector string[12] : strArray[4] = { 2, { "first", "second" } }
#
# the above line defines a vector of strings (each string with maximum length of
# maximum capacity of 4. This vector is initialized with the size 2, and the firs
# in it will be "first" and "second"
#
}

message Message
{
    Complex : complexData   # It will be default initialized.
    Simple  : simpleData = { true, 42 } # Or it can be explicitly initialized
    ref MessageBase : base  # reference to other message defined in the imported
    array int32_t : things[MaxThings] # using a constant value as array size
}
```

Note

FTDL, FCDL and FSDL languages support comments that start with a # and last till the end of line.

Structure

The file has the following basic structure:

FTDL structure.

```
import <import file 1>
import <import file 2>
...
import <import file n>

<type definition 1>
<type definition 2>
...
<type definition n>
```

<import file>	Reference to other ftdl files, without the extension. Types defined in the imported files can also be used later.

<type definition> Can be an `enum`, `struct` or a `message` definition.

Basic Types

The following basic types can be used in the definitions of messages and structs:

- bool
- integers: `int8_t`, `int16_t`, `int32_t`, `int64_t`
- unsigned integers: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`
- floating point numbers: `float32_t`, `float64_t`
- string[<length>]: fixed maximum length character arrays

Note

The default string lenght is 128. You can override this by providing the number next to the string in brackets. Example:
`string[<length>]`

Enumerations

In addition to these basic types enumerations can be defined like this:

Enum Format.

```
enum <enum name>
{
    <enum value name 1> [ = <integer value>]
    <enum value name 2> [ = <integer value>]
    ...
    <enum value name n> [ = <integer value>]
}
```

<enum name> Enum name is a LoLaID

Note

LoLaIDs consist only of alphanumerical characters and should be camel cased. Also, they must not be any of the reserved keywords: `import`, `message`, `struct`, `array`, `vector`, `enum`, `ref`, `instance`, `config`, `interface`, `auto`, `connection`, `layer`, `system`, `in`, `out`, `policy`, `bool`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`, `float32_t`, `float64_t`, `string`

The defined enumerations can be used as a type or they values can be used in places where a number is expected. Enum values can be referenced like this:

```
<enum name>.<enum value name>
```

During generation each `enum` gets an additional value at the end named **MAX_<enum name>**.

Constants

Even though enumerations allow their integer value to be specified, conceptually they are different from constants. In the generated C files they are both represented with `enum`, but in C++, constants are using `constexpr`.

constant format.

```
const <type> : <name> = <value>
```

Structs

These types can used to define messages and structs. Structs can be defined using the following syntax.

Struct Format.

```
struct <struct name>
{
    <typename> : <member name> [ = <initialization>]
    .or.
    array <typename> : <member name>\[<member count>\] [ = <initialization>]
    .or.
    vector <typename> : <member name>\[<member count>\] [ = { <count>, <initializ
    ...
}
```

<struct name>	`struct` names are LoLaIDs
<typename>	can be one of the basic types or an `enum` or `struct` name.
<member count>	A number or an `enum` value reference in [].
<count>	The initial element count of a vector. See the description of vector
<initialization>	A Valid C initialization of the type. There can be no linebreak in the initialization.

Warning

Please note that initializations are not checked by the generator. So the user might get errors during the compilation of the generated code if the initialization is malformed.

All types can also be declared as an **array** or a **vector**.

array	Arrays are fixed sized C arrays of the given type. Translate into `std::array` in C++.
vector	Vectors translate into `struct` of fixed size C array and an `uint32_t` size storing the current size. This comes handy in the generated C++ interface where LoLa::vector is generated, which works mostly like an `std::vector` except it's capacity is fixed.

Messages

Messages are like structs with a few key differences.

- The framework provides allocation mechanism for messages. User must use this to create new messages, instances created manually will possibly cause crashes in the framework.
- Messages are reference counted and automaticcaly deallocated when the last reference is dropped.
- Messages can have references to other messages. Even though circular references are allowed, they are not freed properly, as reference counts will never reach zero.

Message Format.

```
message <message name>
{
    <typename> : <member name> [ = <initialization>]
    .or.
    array <typename> : <member name>\[<member count>\] [ = <initialization>]
    .or.
    vector <typename> : <member name>\[<member count>\] [ = <count>, <initializat
    .or.
    ref <message name> : <member name>
    ...
}
```

<message name>	Message names are LoLaIDs
ref	Declares reference to another message. In C this translates into a message pointer. In C++ this will translate into a LoLa::MessagePtr which works like an std::shared_ptr.

Chapter 6
Outputs

FTDL files

For a messages.ftdl file the generator generates the following files:

C generator	messages.h messages.c
C++ generator	messages.hpp messages.cpp

messages.h contents

1. Necessary includes
2. includes for each import. It's recommended to use the same output folder for all input files or the right Include path has to be set for each folder
3. enum definitions

   Each enum has an additional value generated like MAX_ENUM_NAME

   ```
   enum Numbers
   {
       ZERO,
       ONE = 1,
       TWO,
       THREE,
       MAX_NUMBERS
   }
   ```

4. string conversion functions for each enum

   ```
   const char* <enum name>_to_string(<enum type> value)
   ```

5. struct definitions
6. message definitions

   vectors are generated as anonim structs with a size and a data member

   ```
   typedef struct Message
   {
       struct
   ```

```
        {
            uint32_t size;
            int32_t data[10];
        } sampleVector;
    } Message;
```

7. message creation functions for each message in the form of <message type> createMessageName();

   ```
   SomeMessage* createSomeMessage();
   ```

messages.c contents

1. Include of the respective header
2. Necessary definitions for the framework
3. Implementations for the enum_to_string functions
4. Implementations for the creatMessage functions and function initializations

messages.hpp contents

1. Necessary includes
2. LoLa namespace
3. Enum declarations. To convert the values to string please call the repective C functions.
4. Structure declarations with more convenient C++ types
5. Message declarations with C `types. To create messages a C` message use the LoLa::make_message<LoLa::Message>() function

```
LoLa::MessagePtr<LoLa::MyMessage> msg = LoLa::make_message<LoLa::MyMessage>();
auto msg2 = LoLa::make_message<LoLa::MyMessage>();
```

messages.cpp contents

1. Include of the respective C and C++ message headers.
2. static asserts ensuring the mapping between the C and C++ version of the structures
3. Implementations necessary for make_message to work.

FCDL files

For a components.fcdl file the following file are generated

C generator components.h

	components.c
C++ generator	components.hpp components.cpp

components.h contents

1. Necessary include of message headers
2. For each component
   a. Structure representing a component instance.
   b. Structure representing the parameters that will be passed to the on_create_ComponentName function if it has parameters.
   c. Declaration of the on_create_<component name> function. It has to be implemented by the user.

      ```
      void* on_create_SimpleComponent(const SimpleComponent* me, const SimpleCo
      ```

      or without parameters

      ```
      void* on_create_SimpleComponent(const SimpleComponent* me);
      ```

   d. Declaration of the on_destroy_<component name> function. It has to be implemented by the user.

      ```
      void on_destroy_SimpleComponent(const SimpleComponent* me);
      ```

   e. Message declaration for the synced interface if needed

      ```
      typedef struct SyncedLatest_LatestMessages
      {
          const Message1 *Port1;
          const Message2 *Port2;
          const Message3 *Port3;
      } SyncedLatest_LatestMessages;
      ```

   f. Handle function for the subcribed messages.

      ```
      void handle_SyncedLatest_LatestMessages(SyncedLatest* me, const SyncedLat
      ```

   g. Publish function declarations for each published message.

      ```
      void publish_Source_Output(const Source* me, const Message* message);
      ```

components.c contents

1. Include of the component header
2. Implementation of the publish functions

component.hpp contents

1. Necessary include of the c++ message headers
2. For each compoenent
    a. A class declared for each component that the user can derive from when implementing components.
        i. The handle function is declared pure virtual within the class. The user has to implement them.
        ii. Publish functions are also declared and implemented in the respective source file
    b. Message declaration for the synced interfaces if needed
    c. A class for parameters received during construction of the instances

components.cpp contents

1. Necessary include of the c and c++ component header and message headers.
2. Implementation of the component class

FSDL files

For a system.fsdl file the following files are generated:

C generator	system.c system_typedesc.c if typedesc generation is requested
C++ generator	no files are generated

system.c contents

1. Necessary framwork includes and declarations
2. Preallocated pools for all messages
3. Preallocated instances of components
4. Implementation of dispatching messages for the given system
5. Declarations of the queues and execution contexts used by the system

Chapter 7
C++ interface

Namespaces

Currently everything you have to deal with regarding the `C++` interface is in the `LoLa` (to be renamed very soon to `LoLa`) namespace. This includes both the helper classes like `MessagePtr` and the generated code like the message structures.

In the future this is likely to change, and the namespace will be different for each library or component (to be decided).

> **Warning**
> It's important to note that many of the names are shared between the C and the CPP generated code, with only the namespace being different. Under normal circumstances you should never have to include the C generated code if you are using the CPP one, but if you do, be extra careful to avoid using the wrong type!

Components

The main role of `LoLa` is to transport messages between instances of components. Each component has zero or more incoming ports (subscribing), and zero or more outgoing ports (publishing). For each component there can be any number of instances, with the `FSDL` specifying that which port is connected to which "message network".

In the `C++` code, each component's interface is represented by a base class in the generated code, and its implementation by a user defined class that is inherited from that base class. For each instance the `FSDL` requires to have of that component, there will be one instance of the user defined class. The base class:

- Has zero or one `virtual` function that needs to be overriden by the user's class, this is the function which is called when the component has to process a message (either a synchronized, or a normal message).
- Has zero or more publish functions which the user code can call, these provide a way for the user to publish messages in the name of an instance.
- Has a few utility functions and variables the user can access, and a couple the user is not allowed to access.

Construction

During the initialization of the system, the framework creates zero or more instances of each component. For this it needs to know the type of the class that belongs to the user provided implementation of the component. For this the user has to use the

`LOLA_SET_COMPONENT_INSTANCE_TYPE` macro. It has two parameters: name of the component and type name of the implementation class. See the following example:

simple user code.

```
class MyFooInstance : public LoLa::FooComponent
{
public:
    MyFooInstance(const Parameters&) : LoLa::FooComponent() {}
private:
    virtual void handle_InPortA(LoLa::MessagePtr<const LoLa::TestMessage> message
};

LoLa_SET_COMPONENT_INSTANCE_TYPE(FooComponent, MyFooInstance);
```

The framework will always call the constructor of the user defined component class with a `const` reference to a component specific `Parameters` structure. The members of this structure is defined in the `FCDL` file. It can be empty. Of the component base class, you can call the constructor with or without this parameter.

Messages

Each message type described in the `FTDL` files is represented by a `struct` in the generated `CPP` code. You might allocate message objects directly, but the framework can handle message objects only if they were allocated through it. For the C++ interface this means messages that were:

- Created with the `LoLa::make_message` function
- Created with the `LoLa::copy_message` function
- Received from the framework by subscribing to a message network (see Receiving messages).

For more information please see the documation about `LoLa::MessagePtr`.

Receiving messages

There are two ways a component may be subscribed to messages, depending on whether you want to receive messages from one or from more than one message networks. The latter one is done with synchronized messages.

> **Note**
> A component is not allowed to have more than one direct subscriptions, or more than one synchronized subscriptions, or having both direct and synchronized subscriptions.

Single subscribe

When a component is receiving messages on only one port, it will have a single `subscribe` statement in its **FCDL**:

single subscribe fcdl.

```
component FooComponent
{
    interface
    {
        subscribe TestMessage : InPortA
    }
}
```

The above example means that FooComponent will receive messages with the type `TestMessage` on its incoming port named `InPortA`. For the `CPP` interface this will cause the component's base class to have a pure `virtual` function, which will be called whenever a message arrives to the specified port. The above example generates the following function:

single subscribe function C++.

```
class FooComponent
{
[...]
virtual void handle_InPortA(LoLa::MessagePtr<const LoLa::TestMessage> message)=0;
[...]
};
```

Synchronized subscribe

When a component is receiving messages from more than one ports, it has to use synchronized subscription (remember that components are not allowed to use more than one single subscription).

synchronized subscription fcdl.

```
component FooComponent
{
    interface
    {
        subscribe master : SyncedMessageName
        {
            TestMessage  : InPortA
            SomeMessage  : InPortB[10]
            OtherMessage : InPortC[5]
        }
    }
}
```

On the `CPP` interface this will create a structure that will hold a message pointer to each of the synchronized ports. Depending on the synchronization type these can be allowed to be invalid. A pure `virtual` function receiving a const reference to that structure will be added to the component's base class. For example the above `FCDL` example generates the following structure and function:

synchronized subscription C++.

```
class FooComponent
{
[...]
    struct SyncedMessages1;
    virtual void handle_SyncedMessages1(const SyncedMessageName& message) = 0;
[...]
};

struct FooComponent::SyncedMessageName
{
    LoLa::MessagePtr<const LoLa::TestMessage> InPortA;
    LoLa::MessagePtr<const LoLa::SomeMessage> InPortB;
    LoLa::MessagePtr<const LoLa::OtherMessage> InPortC;
    static constexpr uint32_t queueSize_InPortB = 10;
    static constexpr uint32_t queueSize_InPortC = 5;
};
```

Publishing messages

Components can have zero or more **out ports**. These are declared in the component description, and the system description specifies the message network they are connected to. These ports - just as the message networks they are connected to - are message type specific. In the CPP generated code, the **out ports** are represented by public functions of the base classes of the components. See the following example:

out ports, FCDL.

```
component FooComponent
{
    interface
    {
        publish TestMessage : OutPortA
        publish SomeMessage : OutPortB
    }
}
```

the corresponding generated C++ code:

out ports, C++.

```
class FooComponent
{
[...]
public:
    void publish_OutPortA(LoLa::MessagePtr<const LoLa::TestMessage> message) cons
    void publish_OutPortB(LoLa::MessagePtr<const LoLa::SomeMessage> message) cons
[...]
};
```

Chapter 8
lola::MessagePtr

Overview

`lola::MessagePtr` is a smart pointer for message objects created / managed by lola. It's behaviour is dependent on whether it points to a const, or a non-const message object:

- For const message objects, it can have multiple owners; acts like a `std::shared_ptr`.
- For non-const message objects, there can be only one owner; acts like a `std::unique_ptr`.

Although you could create instances of the lola message structures without the help of the framwork, for using the framework you need to have created the message through or received from the framework. The framework enforces this with only accepting messages stored in **`lola::MessagePtr`** objects; and you cannot store (at least not with anything short of hacking with reinterpet_cast) message objects in **`lola::MessagePtr`**-s unless those message objects are store already in an other **`lola::MessagePtr`**.

Creating messages

There are two ways you could create messages (which are useable by the Framework):

1. Create a new message, this is done with the `lola::make_message` function.
2. Copy an existing message, this is done with the `lola::copy_message` function.

lola::make_message

```
template<class MessageType> MessagePtr<MessageType> make_message();
```

`lola::make_message` creates a new message. It initializes its members with the values specified in the FTDL.

lola::copy_message

`lola::copy_message` takes an existing message and creates a copy of it. The returned message is guaranteed to be owned only by the caller, and it will be a pointer to a non-const message object. If you call **`lola::copy_message`** with an invalid **`lola::MessagePtr`**, it returns an invalid **`lola::MessagePtr`**.

> **Warning**
> It is very important to remember, that copying the **`lola::MessagePtr`** object (only allowed for const messages) is not copying the message, only the pointer.

```
void Foo(lola::MessagePtr<TestMessage> message)
{
// creates a new TestMessage object, and copy the contents of message into it.
    lola::MessagePtr<TestMessage> myMessage = lola::copy_message(message);
    myMessage->someVariable = myMessage->someVariable + 1;
    std::assert(myMessage->someVariable == message->someVariable + 1);
}
```

Note

`lola::copy_message` is optimized to take advantage of move semantics if possible: if there are no other owners of the message object then the **`lola::MessagePtr`** that got moved into the **`lola::copy_message`** function, then **`lola::copy_message`** will not copy the contents of the message:

```
void Foo()
{
    lola::MessagePtr<const TestMessage> myMessage = lola::make_message<const Test
    lola::MessagePtr<const TestMessage> anotherReferenceToMessage = myMessage;
//
// at this point there are two owners of the message:
// myMessage and anotherReferenceToMessage both own it.
//
    {
        lola::MessagePtr<const TestMessage> otherMessage =
            lola::copy_message(std::move(myMessage));
//
// the above call does create a new message and copies the contents of
// myMessage into it, because myMessage was not the single owner of the
// TestMessage object it points to. (anotherReferenceToMessage owns it too)
//
        assert(myMessage.empty()); // ptr (and ownership) got moved out from it.
//
// at this point there are two owners of the message:
// otherMessage and anotherReferenceToMessage. myMessage no longer owns it, becau
// it lost when it transfered it to the parameter of lola::copy_message
//
    }
//
// at the end of the above block otherMessage's scope ended, so it was destroyed
// now only anotherReferenceToMessage has ownership of the message.
//
    lola::MessagePtr<const TestMessage> otherMessage
        = lola::copy_message(std::move(anotherReferenceToMessage));
//
// the above call does not create a new message, because myMessage was the
// single owner of the TestMessage object it points to.
//
}
```

Using lola::MessagePtr

It's best to think about **`lola::MessagePtr<T>`** as if it was `T*`. You can dereference it, access members of `T` with operator `->`, check for validity, compare two of them, copy it,

etc… And for all those operations you need to keep in mind, that you are operating on the pointer, and not on its contents.

Examine the following example:

```
bool CompareMessages(lola::MessagePtr<const TestMessage> a,
    lola::MessagePtr<const TestMessage> b)
{
    if (a==b)  // this compares the values of the pointers,
               // not the objects they point to
    {
        return true;
    }
    else
    {
        if (a&&b) // checking if both are valid
                  // equivalent to !a.empty()&&!b.empty()
        {
            return *a==*b; // comparing the objects a and b point to.
        }
        return false;
    }
}
```

`lola::MessagePtr` for non-const messages are move-only objects. Copy assignment and copy construction is not allowed.

`lola::MessagePtr` for const messages supports move semantics, and it is recommended that you use that whenever possible, because copying **`lola::MessagePtr`**-s for const messages means sharing ownership of the message object between the copies of the **`lola::MessagePtr`**-s, and ownership sharing is handled with reference counting, which has a performance impact.

`lola::MessagePtr` may point to either const or non-const message object, operations between those are like what you could except from pointers to const and non-const objects:

```
void Foo(lola::MessagePtr<const TestMessage> myConstMessage,
    lola::MessagePtr<TestMessage> myMessage)
{
    myMessage = myConstMessage;
    // COMPILE ERROR

    myConstMessage = myMessage;
    // works

    bool result = myMessage==myConstMessage;
    // works, you can compare const and non-const pointers

    bool result2 = myMessage && myConstMessage && *myMessage==*myConstMessage;
    // works, comparing the contents of the TestMessage objects.

    myMessage = lola::copy_message(myConstMessage);
    // works, creating a new copy from const TestMessage object.
```

```
}
```

Just as a regular pointer, **`lola::MessagePtr`** might be invalid (equivalent to a regular pointer having the value of nullptr). Its this invalid state the only state you can create a **`lola::MessagePtr`** object without the help of the framework. If you have a **`lola::MessagePtr`** pointing to a message object, you can release its ownership of the message object and set its state to invalid with the reset call:

```
void Foo(lola::MessagePtr<TestMessage> message, lola::MessagePtr<TestMessage> mes
{
    message.reset();    // recommended, "clearer" way
    assert(message.empty());
    message2 = nullptr; // this works also, because lola::MessagePtr
                        // can be constructed from std::nullptr_t
    assert(message2.empty());
}
```

Chapter 9
lola::String

Overview

lola::String is basically a wrapper for char[] type of strings, with fixed max length, and zero terminated, but (and this is a very important) only if the string has less than the maximum-length length). Its primary use is intended to be as a member in LoLa messages, and for that its memory footprint is kept as the exact same as what messages of LoLa's C interface use.

One important thing to note is that lola::String does not store the current length of the string, and because of this any operation that needs that info has a minimum linear (in size) time complexity.

> **Warning**
> Always keep in mind that if the stored string's length is the same as the maximum size of the containing String, then the string is not zero terminated! This means that for example the following code has undefined behavior:

```
void Foo(const lola::String  &myString)
{
    printf("%s", myString.c_str()); // undefined behavior when myString.size()==m
}
```

instead, you could write:

```
void Foo(const lola::String  &myString)
{
    printf("%.*s", myString.size(), myString.c_str()); // correct code
}
```

Functions

The interface of lola::String mimics the interface of std::string where possible. There are a few important differences though:

- Because the maximum length of lola::String is fixed, operations that grow the string's length may fail, for example push_back.
- npos is a function, not a static constant. The reason for this is the template argument of lola::String would make it cumbersome to use.
- c_str does not guarantee returning zero terminated string, when size()==max_size() the returned const char* string won't be zero terminated!

- the performance of the functions that need to know the length of the string are linear with the length of the string. (so for example lola::String::length()'s complexity is linear in length; std::string::length() is constant (C++11))
- You can convert lola::String to std::string with lola::String::to_string function.
- You can compare (only equality) lola::String with std::string and const char* strings.
- Although std::hash is available for lola::String, currently it’s performance is linear in length, thus not really recommended for usage.

Chapter 10
Sync policies

Overview

When a component receives messages from more than one port, it is not allowed to process the messages from those port independent of each other. Instead it has to specify a policy that will be used for synchronizing the messages between these ports. What this means is that whenever a message is received on one of these ports, that message is not received directly by the user, but by the framework. The framework maintains the synchronization state (specific for the selected policy) for each instance. Based on that state the received message will either generate a *SyncedMessage*, which is passed to the user, or it will change the state, but stay hidden from the user. *SyncedMessages* contain references to a single message from all synchronized ports, the used policy decides which message will be referenced.

Note

When using a message synchronizing policy, each incoming message generates zero or one *SyncedMessage*.

Note

A component is allowed to have only one synchronized group of ports. This means that if you have more than one incoming port, you have to put all incoming ports into one synchronized message group, and select one policy for those.

Monotonic increasing timestamps

All sync policy requires that for each port, the timestamp of messages are monotonic increasing. Any message that violates it is ignored. Between different ports there are no such requirments. Repeating the same timestamp is allowed.

When there’s only one source sending messages to a port, this requirment shouldn’t cause concerns, but when there are multiple sources sending messages to the same network, thus the same port, it is something that has to be kept in mind.

FCDL format

FCDL format
subscribe <policy name> : <name of synchronized message>
{
<message type 1> : <port name 1>[[queue length 1]]
<message type 2> : <port name 2>[[queue length 2]]
…
}

policy name	Name of the synchronization policy. Can be "latest", "master", or "timed".
name of synchronized message	Name of the message that the user code will receive.
message type	Type of the message to be expected from the port specified in the same line.
port name	Name of the port. Must be unique within the component.
queue length	Timed policy requires this parameter for all of its ports, Master policy requires this for all but the first port. For more information about its meaning check those policies.

Example synched subscription.

```
component Foo
{
    interface
    {
        subscribe master : MySyncedMessage
        {
            SomeMessage  : InPortA
            OtherMessage : InPortB[5]
            SomeMessage  : InPortC[3]
        }
    }
}
```

About the examples

This document provides many examples, and they all follow the same format:

Example 10.1. example sample

Msg #1: PortA, time 100
Msg #2: PortB, time 100
SyncedMessage: #1, #2
Msg #3: PortA, time 110

SyncedMessage: #3, #2
Msg #4: PortB, time 120
Note: there is no SyncedMessage generated for Msg #4
Msg #5: PortA, time 150
SyncedMessage: #5, #4

In each example, there are three types of lines:

- Lines that begin with **Msg** #, these represent messages arriving to one of the incoming ports of the component. The line's format is the following: `Msg #<message id>: <port name>, time <timestamp of message>`. You always have to assume that "some time" has elapsed since whatever was the previous event.

- Lines that begin with **SyncedMessage**, these represent the *SyncedMessages* generated by the framework. The previous line always must be a line with a message arriving to an incoming port. The *SyncedMessage* is generated in response to that. The line's format is the following: `SyncedMessage: <message id for PortA> <message id for PortB> [,message id for PortC]`.

- Notes. These begin with the word "*Note*", and are italic. Some long notes might extend to multiple lines.

In all of the examples, the first incoming port is PortA, the second is PortB, and in those that has 3 ports, the third one is PortC.

Latest policy

Latest is the simplest sync policy. It generates a *SyncedMessage* whenever a message arrives to any of the synchronized ports, and the *SyncedMessage* will hold a reference to the latest message received on each port (of course, one of these is the message that triggered the SyncMessage). No *SyncedMessage* is generated until all of the synchronized ports received at least one message.

Example 10.2. Latest policy example

Note: this component has 2 ports
Msg #1: PortA, time 100
Note: no SyncedMessage is generated, because PortB has not received a message yet.
Msg #2: PortB, time 120
SyncedMessage: #1, #2
Msg #3: PortB, time 130
SyncedMessage: #1, #3
Msg #4: PortA, time 105
SyncedMessage: #4, #3

> **Note**
> if a message violates the monotonic increasing timestamp policy, that message is ignored. No *SyncedMessage* is generated for such message.

Msg #1: PortA, time 100
Msg #2: PortB, time 120
SyncedMessage: #1, #2
Msg #3: PortB, time 130
SyncedMessage: #1, #3
Msg #4: PortB, time 125
Note: Msg #4's timestamp is lower than #3, which is for the same port. Because of this #4 is ignored.
Msg #5: PortA, time 110
SyncedMessage: #5, #3
Note: Msg #5's timestamp was lower than #4's, but that doesn't matter because they are for different ports.

Master policy

While with the **Latest policy** treats all of its ports equally, the **Master policy** has one *master port*, and one or more *slave ports*. It generates *SyncedMessages* only if a message is received by its *master port*, messages arriving to the *slave ports* never generate *SyncedMessages*.

If a *slave port* has not received any message yet, the pointer to the message for that port will be `NULL`.

*Note: In the examples for **Master** policy, PortA is the master port, PortB and PortC are slave ports*
Msg #1: PortA, time 100
SyncedMessage: #1, NULL, NULL
Msg #2: PortB, time 120
Msg #3: PortA, time 110
SyncedMessage: #3, #2, NULL
Msg #4: PortC, time 130
Msg #5: PortA, time 120
SyncedMessage: #5, #2, #4

For each of its *slave ports*, it stores the last x messages, where x is defined per *slave port* in the `FCDL`. When a message arrives on the *master port*, for each of the *slave ports* it looks up the message with the timestamp closest to the timestamp of the *master port's* message, and uses that for the SyncedMessage. In case of a tie, the newer message is selected.

Note: assume that PortB has a queue length of 3
Msg #1: PortC, time 100
Msg #2: PortB, time 100
Msg #4: PortB, time 110
Msg #5: PortB, time 120
Msg #6: PortB, time 130
Note: Msg #2 is discarded at this point (PortB queue length = 3)
Msg #7: PortA, time 100
SyncedMessage: #7, #4, #1

Note: of the queued messages for PortB, #4 was the nearest to time 100.

Timed policy

Timed policy is the most complex of the synced policies:

- It has a primary port, and one or more secondary port(s).
- It has a predefined *tolerance* value, which is a time interval.
- *SyncedMessages* constructed in a way that within a *SyncedMessage*, the timestamp for the messages for the secondary ports are within *tolerance* distance from the timestamp of the message for the primary port. Note that this means that between two secondary port message the maximum time difference is twice the *tolerance* value.
- *Tolerance* can be 0, which means that the timestamps of the messages in the *SyncedMessage* have to be the same.
- Each of its ports has a predefined queue length, meaning that maximum that many messages may be stored for that port.
- Monotonic increasing timestamp policy: if a message arrives with lower timestamp than a previous message for *that* port, this new message is ignored.
- No synced message will ever have a message that has lower timestamp than a previous synced message had that for that port.
- Whenever a new message arrives, the new message is stored in the queue for its port. Then a synced message is generated:
  - consider all possible *SyncedMessages* allowed by the already listed rules, for which it is true that the new message is present in it. If no such message is possible, no SyncedMessage will be generated.
  - of these possible *SyncedMessages*, select the one where the timestamp of the primary port's message is the highest (most recent).
  - for the secondary ports, select the messages with the highest timestamp allowed (they must be within *tolerance* distance from the message for the primary port).
- If a message arrives with a timestamp that is equal to the highest timestamp among its stored messages for its port, it replaces that message (and then it checks if it can generate a *SyncedMessage* with the just arrived message)
- Messages are not "used up" with the *SyncedMessages*, a message may be present in multiple *SyncedMessage*

Examples

For all examples for the **timed policy**, PortA is the primary port.

Number of ports: 3, Tolerance: 0
Msg #1: PortA, time 100
Msg #2: PortB, time 100
Msg #3: PortC, time 100
SyncedMessage: #1, #2, #3
Msg #4: PortB, time 100
SyncedMessage: #1, #4, #3
Note: messages are not used up when a SyncedMessage is generated
Msg #5: PortC, time 110
Msg #6: PortC, time 110
Note: #6 replaced #5
Msg #7: PortB, time 110
Msg #8: PortA, time 110
SyncedMessage: #8, #7, #6

When there's only 2 ports, there's no difference in behaviour between the primary and the secondary port. Whenever a message arrives to one of them, the framework looks for a message from the other port that has a timestamp within tolerance from the just arrived message. If there are more than one such messages, it selects the one with the highest timestamp:

Number of ports: 2, Tolerance: 10
Msg #1: PortA, time 200
Msg #2: PortA, time 205
Msg #3: PortA, time 210
Msg #4: PortB, time 189
Msg #5: PortB, time 190
SyncedMessage: #1, #5
Msg #6: PortB, time 195
SyncedMessage: #2, #6
Msg #7: PortB, time 200
SyncedMessage: #3, #7
Msg #8: PortB, time 210
SyncedMessage: #3, #8
Msg #9: PortB, time 221

It's a somewhat similar scenario, when there are more than 2 ports, and a message arrive to the primary port. For each secondary port, the framework finds the best match (highest timestamp within limit). If there is a secondary port for which it could not find a message, then there is no SyncMessage generated:

Number of ports: 3, Tolerance: 10
Msg #1: PortB, time 200
Msg #2: PortB, time 205
Msg #3: PortB, time 210
Msg #4: PortC, time 202
Msg #5: PortC, time 207
Msg #6: PortC, time 212

Msg #7: PortA, time 191
Note: PortB has a valid candidate, but there's no message queued for PortC within tolerance limits
Msg #8: PortA, time 192
SyncedMessage: #8, #1, #4
Msg #9: PortA, time 199
SyncedMessage: #9, #2, #5
Msg #10: PortA, time 200
SyncedMessage: #10, #3, #5
Msg #11: PortA, time 202
SyncedMessage: #11, #3, #6
Msg #12: PortA, time 221

When there are more than 2 ports, and a message arrives to one of the secondary ports, it's very important to remember that the framework's first priority is maximizing the timestamp of the primary port. And that the timestamps of the messages from the other secondary ports are compared to the primary port's message's timestamp, not the just arrived message's.

Number of ports: 3, Tolerance: 10
Msg #1: PortA, time 200
Msg #2: PortA, time 205
Msg #3: PortA, time 210
Msg #4: PortB, time 194
Msg #5: PortC, time 210
SyncedMessage: #1, #4, #5
Note: all 3 enqueued messages were within tolerance limit from msg #5, but only #1 had a message within tolerance limit from portB

Chapter 11
lola::Vector

Overview

lola::Vector is a fixed maximum capacity vector (dynamic sized array). Its interface tries to mimic the interface of std::vector. Its primary use is intended to be as a member in LoLa messages, and for that its memory footprint is kept as the exact same as what messages of LoLa's C interface use.

Noteable differences from std::vector:

- Because lola::Vector's capacity is fixed:
  - There is no reserve function, and capacity always returns the same value.
  - Any function that (might) increases the size (push_back, resize, insert, …) may fail.
  - lola::Vector-s with different capacities (MaxSize template argument) are considered different type even if the stored type is the same.
  - lola::Vector does not allocate memory dynamically, so it is much more like std::array than std::vector in that regard.
- Moving and copying has the same speed (linear in size()).
- lola::Vector has no space optimized specialization for bool type.
- The "at" function does not throw out_of_range exception
- There's a convenience function const_iterator_to_iterator, which you can use to convert const_iterator to iterator.
- resize leaves elements of the array uninitialized if T is trivial (std::is_trivial<T>)

End-User License Agreement with Embedded Software Supplement

The latest version of the End-User License Agreement with Embedded Software Supplement is available on-line at: www.mentor.com/embeddededeula.

IMPORTANT INFORMATION

USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDER, FEES AND PAYMENT.**

   1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.

   1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.

   1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer'sprovision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited,

for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of MentorGraphics.

3. **BETA CODE.**

3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively "Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.

3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of thisAgreement.

4. **RESTRICTIONS ON USE.**

4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer's product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively "Files"), including without limitation files containing Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") which are Mentor Graphics' trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.

4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer's employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.

4.3. Customer agrees that it will not subject any Product to any open source software ("OSS") license that conflicts with this Agreement or that does not otherwise apply to such Product.

4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successorsin interest and assigns.

4.5. The provisions of this Section 4 shall survive the termination of this Agreement.

5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics' then current End-User Support Terms located at http://supportnet.mentor.com/supportterms.

6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations ("Third Party Terms") may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement (including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third partycode. The provisions of this Section 6 shall survive the termination of this Agreement.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **THIRD PARTY CLAIMS.**

9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.

9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.

9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

10. **Infringement.**

10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of theaction.

10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.

10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE, SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THISAGREEMENT.

11. **TERMINATION AND EFFECT OF TERMINATION.**

11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.

11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.

12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.

13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.

14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting

firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.

16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the Stateof Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.

Rev. 151102, Part No. 267028 (with ESS)

EMBEDDED SOFTWARE SUPPLEMENT

This Embedded Software Supplement (“Supplement”) is attached to and made a part of Mentor Graphics End-User License Agreement, which may be viewed at www.mentor.com/eula. ("Agreement"), and governs the licensing of Embedded Software to Customer by the Mentor Graphics entity that issued the corresponding quotation (“Mentor Graphics”). Any conflict between the terms of this Supplement and those of the Agreement as it relates to Customer's use of Embedded Software shall be resolved in favor of this Supplement.

1. **Introduction.** Unless otherwise noted, the capitalized terms used in this Supplement shall have the same meanings as set forth in the Agreement.

2. **Definitions.** As used in this Supplement and the applicable quotation (“Quotation”), these terms shall have the following meanings:

   2.1. “Customer's Product” means Customer's product identified by a unique stock keeping unit or “SKU” in the Quotation that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to Customer's customer, whether End-Users or intermediate manufacturers;

   2.2. “Developer” means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the Embedded Software in source code form, or creates, modifies or compiles Software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product;

   2.3. “Development Location” means the location where Software may be used as authorized in the Quotation;

   2.4. “Development Tools” means the Software that may be used by Customer for creating, editing, compiling, debugging or prototyping Customer's Product;

   2.5. “Embedded Software” means Software that is embeddable;

   2.6. “End-User” means consumers that receive Customer's Products after such products have been fully developed and marketed;

   2.7. “Executable Code” means a compiled program translated into a machine-readable format that can be loaded into memory and run by a Processor;

   2.8. “Linkable Object Code” or “Object Code” means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;

   2.9. “Mentor Embedded Linux” or “MEL” means Mentor Graphics' tools, Source Code, and recipes for building Linux systems;

   2.10. “Open Source Software” or “OSS” means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;

   2.11. “Processor” means the specific microprocessor to be used with Software and implemented in Customer's Product;

   2.12. “Proprietary Components” means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an OSS license, as more fully set forth in the product documentation provided with the Products;

   2.13. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer's Linkable Object Code developed with Software, as more fully set forth in the documentation provided with the Products;

   2.14. “Software” means Mentor Graphics software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Supplement;

   2.15. “Source Code” means software in a form in which the program logic is readily understandable by a humanbeing;

   2.16. “Sourcery CodeBench Software” means Mentor Graphics' Development Tool for C/C++ embedded application development;

   2.17. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors; and

2.18. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary's use of Products under this Supplement, Customer agrees to ensure such Subsidiary's compliance with the terms of this Supplement, the Quotation and the Agreement and will be liable for any breach by a Subsidiary.

3. **Grant of License.** Subject to the payment of the applicable license fees, Mentor Graphics grants to Customer, a nontransferable, nonexclusive license to use Software as described in the Quotation or, if no license is specified in the Quotation, Mentor Graphics grants to Customer a nontransferable, nonexclusive license to use Software solely for Customer's internal business purposes. The limited licenses granted under the Quotation or this Supplement shall continue until the expiration date of term-licensed Products or termination in accordance with the termination provision of the Agreement, whichever occurs first. Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software, beyond the scope specified in the Quotation or this Supplement.

4. **Support Services.**

4.1. Except as described in Sections 4.2, 4.3 and 4.4 below, and unless otherwise specified in the Quotation, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then- current End-UserSupport Terms located at http://supportnet.mentor.com/supportterms.

4.2. To the extent Customer purchases support services for Sourcery CodeBench Software, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at http://www.mentor.com/codebench-support-legal.

4.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at http://www.mentor.com/vsipl-support-legal.

4.4. To the extent Customer purchases support services for MEL, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current MEL Support Terms located at http://www.mentor.com/terms_conditions/mel-software-license. http://www.mentor.com/terms_conditions/mel-software-license.

5. **Effect of Termination.** In the event of termination, Customer shall discontinue its use of the Product in accordance with the termination provisions of the Agreement, except to the extent that an Open Source Software license permits Customer's continued use of any Open Source Software portion or component of a Product. Further, upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product, provided that: (a) the End-User was licensed according to the terms of this Supplement or the Agreement, if applicable to such End-User; and (b) such End-User is not in breach of its agreement with Customer or Mentor Graphics, if applicable, nor a party to Customer's breach.

6. **Open Source Software.** Customer agrees that it will not subject any Product provided by Mentor Graphics to any OSS license that does not otherwise apply to such Product. Products may contain OSS or code distributed under a proprietary license agreement to which additional rights or obligations (“Third Party Terms”) may apply. Please see applicable software product documentation, including but not limited to license notice files, header files or source code for further details. In the event of conflict between the terms of this Supplement, the Quotation, the Agreement and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of the Agreement and this Supplement.