User Guide for UNIFI Software-in-the-Loop (SIL) Wrapper SIMULINK Library

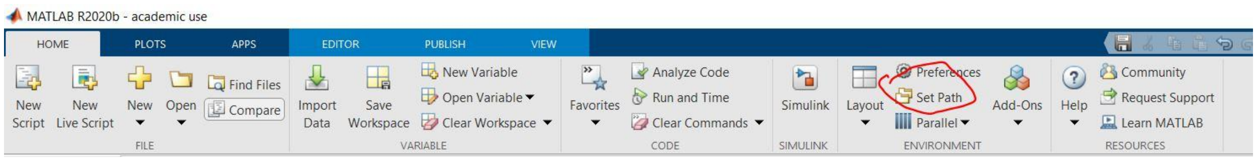
Author M A Awal
Senior Control Engineer, Medium Voltage R&D, Danfoss Drives
Adjunct Assistant Professor, North Carolina State University

Contributors Md Rifat Kaisar Rachi
PhD Candidate, FREEDM Systems Center, North Carolina State University

1. Library Installation

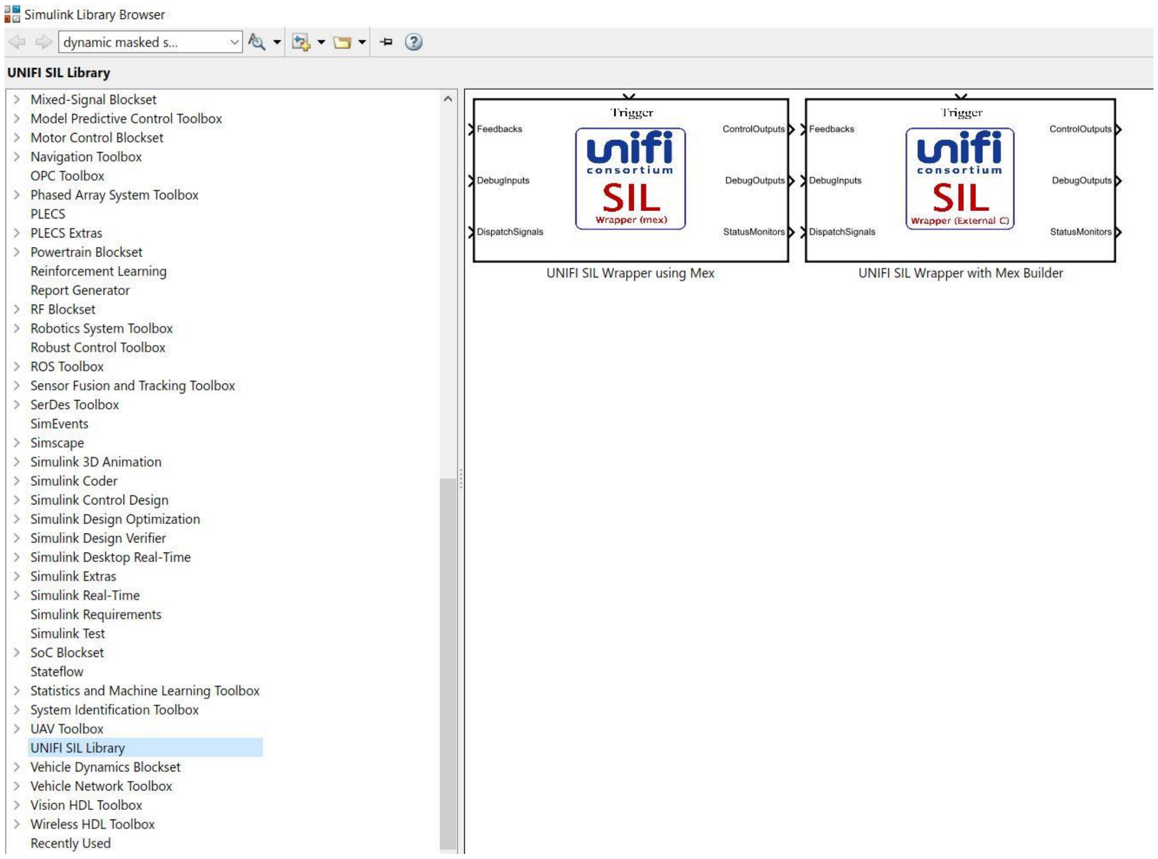
i. Unzip the library files into a folder of your choice on your computer. Use standard naming convention for the folder, such as 'no space in file path,' as per required by MATLAB/SIMULINK.

ii. Add the library folder to MATLAB path.

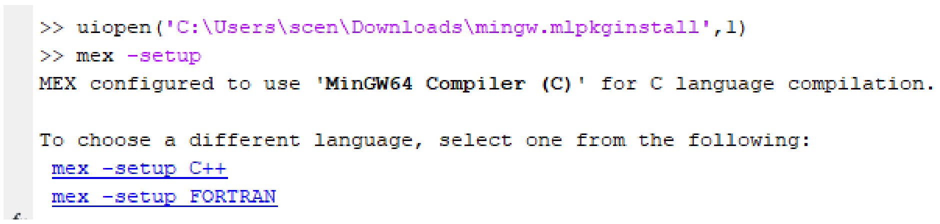


iii. Open Simulink → New Model → Library Browser → Refresh Library Browser (right-click on any library and select "Refresh Library Browser"/ shortcut - F5)

iv. The *UNIFI SIL Library* should be available in the list of libraries.



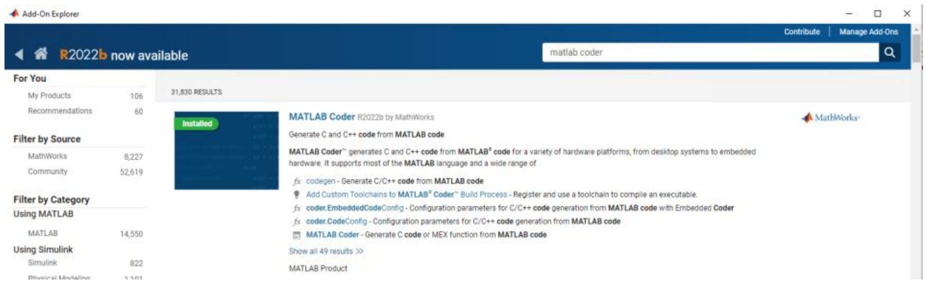
v. For wrapping control software written in C/C++, install MinGW-w64 C/C++ compiler support in MATLAB. Detail instructions for installation can be found here (Link). Once installed, to verify mex (MATLAB executable) compiler configuration, type *mex –setup* in MATLAB command window; if configured properly, "*MEX configured to use 'MinGW64 Compiler (C)' for C language compilation*" should be returned by MATLAB.



```
>> uiopen('C:\Users\scen\Downloads\mingw.mlpkginstall',1)
>> mex -setup
MEX configured to use 'MinGW64 Compiler (C)' for C language compilation.

To choose a different language, select one from the following:
 mex -setup C++
 mex -setup FORTRAN
```

vi. Install MATLAB Coder Add on.



31,830 RESULTS

MATLAB Coder R2022b by MathWorks

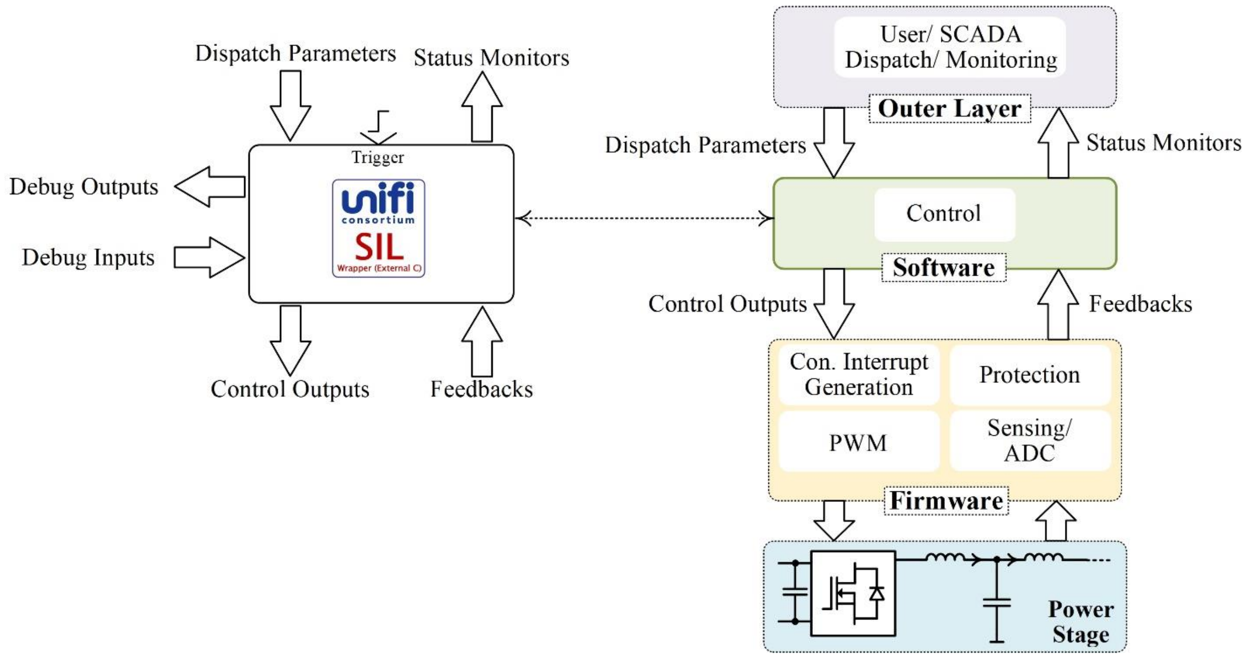
Generate C and C++ code from MATLAB code

MATLAB Coder™ generates C and C++ code from MATLAB® code for a variety of hardware platforms, from desktop systems to embedded hardware. It supports most of the MATLAB language and a wide range of

- codegen - Generate C/C++ code from MATLAB code
- Add Custom Toolchains to MATLAB® Coder™ Build Process - Register and use a toolchain to compile an executable.
- coder.EmbeddedCodeConfig - Configuration parameters for C/C++ code generation from MATLAB code with Embedded Coder
- coder.CodeConfig - Configuration parameters for C/C++ code generation from MATLAB code
- MATLAB Coder - Generate C code or MEX function from MATLAB code

2. Wrapper Block - *UNIFI SIL Wrapper with Mex Builder*

The *UNIFI SIL Wrapper with Mex Builder* is used for linking external C code-base/library in SIMULINK environment. An example of a typical power electronics converter is shown in Fig. 3.



The overall control structure can be categorized into three layers –

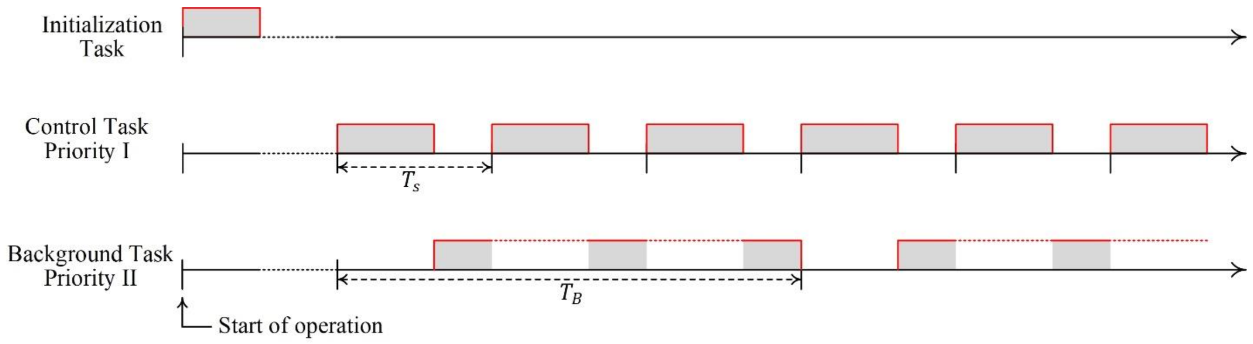
a. **Firmware**: sensing, analog-to-digital conversion (ADC), pulse-width-modulation (PWM), sub-PWM cycle time-scale protection, and control-update-interrupt generation etc., are 'control hardware specific'. For instance, each microcontroller (uC)/digital signal processor (DSP) manufacturer, such as Texas Instruments/ST/Xilinx, requires platform specific codes for configuring/accessing the dedicated peripherals, e.g., ADC/PWMs.

b. **Software**: In this context, ‘software’ is defined as ‘control hardware independent’ portion of the code, ideally which should be portable across different control hardware platforms.
c. **Outer Layer**: For any power electronics application of substantial size, typically a user interface and/or a SCADA interface is needed for dispatching operational parameters/set-points and/or monitoring operating conditions.

The UNIFI SIL Wrapper environment can be used to develop new control **Software** or to evaluate existing **Software**, written in C. Embedded code, e.g., written for TI DSP or ST uC, should be structured/segmented into **Software** and **Firmware** parts to enable the use of UNIFI SIL Wrapper environment.

2.1. Task Groups

Note that only the software part of the code is to be run by the UNIFI SIL block. The software is grouped into three parts, which correspond to the typical control execution in a power electronics application, shown as follows:



Initialization Task: Initialization of controller states and configuration of parameters etc. done at the start of operation.

Control Task: Highest priority tasks updated at the sampling rate $f_s = 1/T_s$. Typically current/voltage reference tracking loops, droop controllers etc. are included in this task group.

Background Task: Lower-priority tasks, such as updating SCADA dispatched parameters/power set-points, sending status updates to SCADA, are updated at a slower rate T_B.

The UNIFI SIL block has provisions for all three task groups. The user may choose to use any or all three task groups.

2.2. I/O Structures

The SIL block allows three groups of I/Os –

Feedbacks & Control Outputs: The *feedbacks* (input) structure allows to pass signal values from SIMULINK to the *software* and the *control outputs* (output) structure allows SIMULINK environment to receive values from the *software*. Both structures are updated (data transferred) at the sampling rate, $f_s = 1/T_s$.

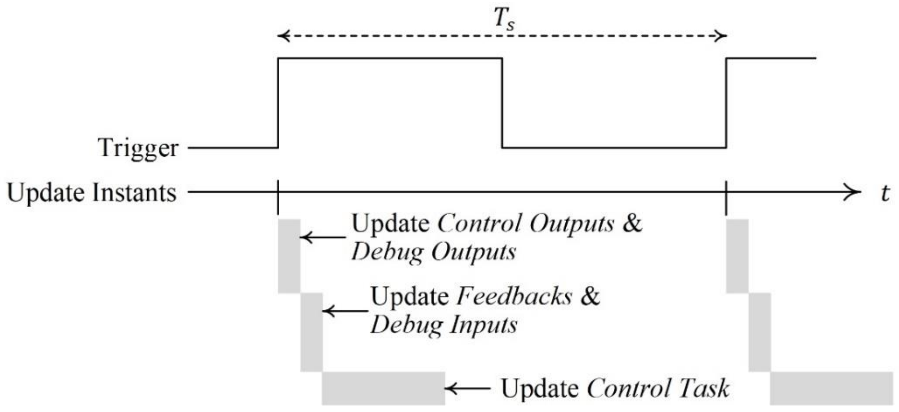
Dispatch Parameters & Status Monitors: The *dispatch parameters* (input) structure allows to pass signal values from SIMULINK to the *software* and the *status monitors* (output) structure allows SIMULINK environment to receive values from the *software*. Both structures are updated (data transferred) at the background task rate, $f_B = 1/T_B$.

Debug Inputs & Debug Outputs: The debug I/O structures can be used to vary/ probe internal controllers signals and/or parameters for debugging purposes. The *debug inputs* (input) structure allows to pass signal values from SIMULINK to the *software* and the *debug outputs* (output) structure allows SIMULINK environment to receive values from the *software*. Both structures are updated (data transferred) at the sampling rate, $f_s = 1/T_s$.

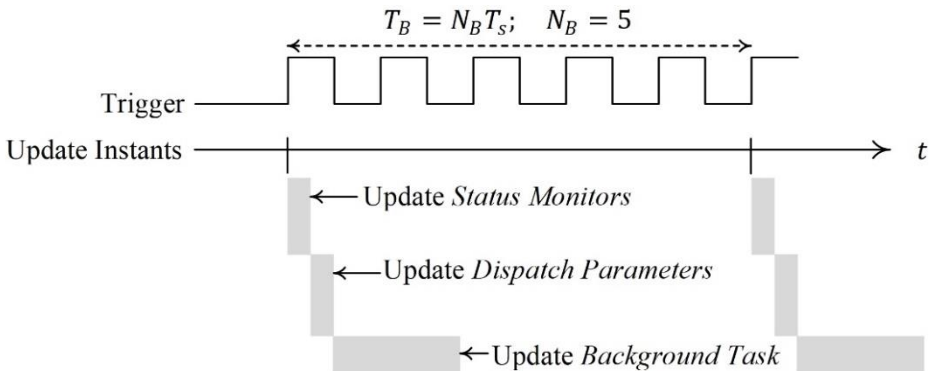
Note that all I/O structures use 32-bit floating point type data.

2.3. Update Trigger & Order of Execution

The *initialization task* is executed only once at the beginning of the simulation. The SIL block is updated in a discrete fashion and the update instants are defined by the rising edge of the *trigger* input. Use the SIMULINK *pulse generator* block with amplitude of 1 for the update *trigger*. The update of the I/O structures and the control and background tasks are sequenced in such a way to emulate the implementation delay by a digital control hardware. The control task and its associated I/O structure update sequence is shown in Fig. 5. Note that all updates are performed immediately at the update instant (at the rising edge of the trigger signal); the grey boxes are exaggerated to denote only the logical sequence of execution at the update instant.


T_s
Trigger
Update Instants
t
Update *Control Outputs* & *Debug Outputs*
Update *Feedbacks* & *Debug Inputs*
Update *Control Task*

Due to the update sequence of the *Control Outputs, feedbacks,* and the *Control Task,* one-sample delay is emulated in the closed-loop control controller using *feedbacks* as input and passing *control outputs* as the output. The background task is updated at $f_B = 1/T_B$ rate where $T_B = N_B T_s$ (see Fig. 6). $N_B = 5$ is used as default value.

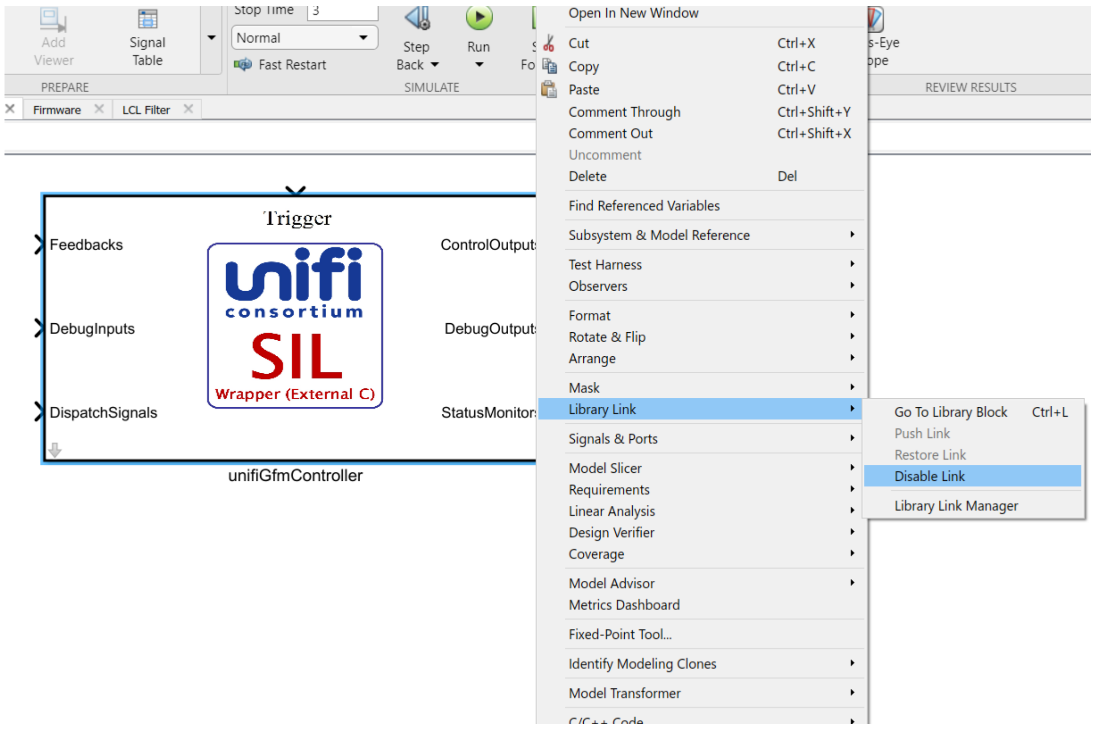


2.4. Block Setup Steps

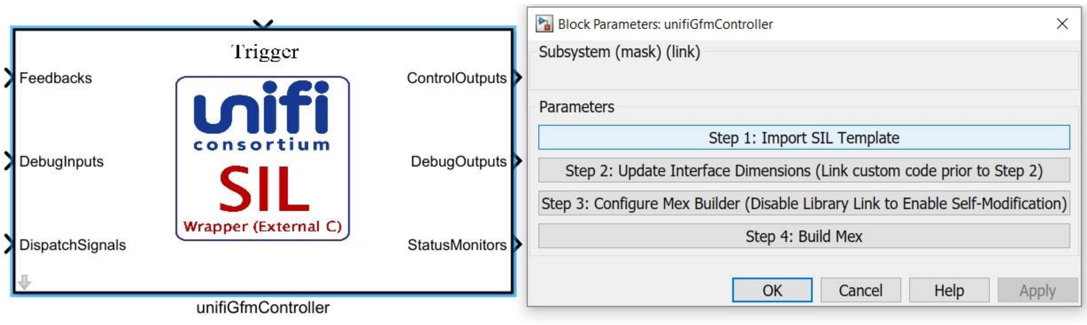
i. Drag and drop the library block *UNIFI SIL Wrapper with Mex Builder* into your SIMULINK model. Save the model as *.slx file in a folder/directory of your choice and change the MATLAB working directory to the same folder.

ii. Name the SIL block without any space in the name; avoid special characters; avoid numbers as the first letter. For this example, we choose the name *unifiGfmController.*

iii. The UNIFI SIL Wrapper blocks are self-modifying blocks and hence the library link must be severed prior to the build steps. Right click the SIL block → Library Link → Disable Link



iv. In the *Block Parameters* window (double click on the SIL block), the first tab *Step 1: Import SIL Template* creates a folder named *wrapperBuildDir* in the working directory and imports the necessary scripts for building the MATLAB executable (mex) file which is used during simulation. Note that all script names start with the name you chose for the SIL block.

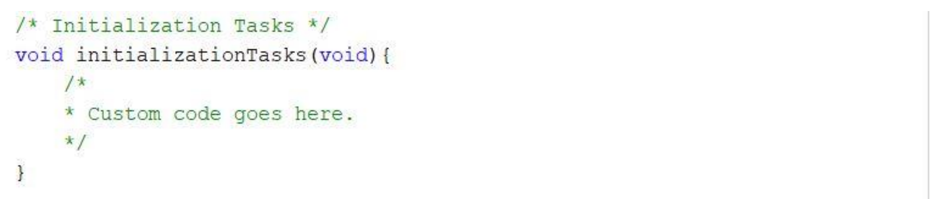

Trigger
Feedbacks
DebugInputs
DispatchSignals
ControlOutputs
DebugOutputs
StatusMonitors
unifi consortium SIL Wrapper (External C)
unifiGfmController
Block Parameters: unifiGfmController
Subsystem (mask) (link)
Parameters
Step 1: Import SIL Template
Step 2: Update Interface Dimensions (Link custom code prior to Step 2)
Step 3: Configure Mex Builder (Disable Library Link to Enable Self-Modification)
Step 4: Build Mex
OK
Cancel
Help
Apply

v. Prior to step 2, link the external C code to the SIL wrapper block. First, open the blockName_interface.h file (unifiGfmController_interface.h in this example). The I/O structures, such as *Feedbacks, Dispatch Parameters, Control Outputs,* and *Status Monitors*, are defined here. Add/remove/edit the variable/members of structures as needed; do not alter the names of the structures. Note that the variables/members of the *debug input* and *debug output* structures are not defined, only the number of variables in each structure in defined. Change the number of debug input and output variables as needed using the macro values

---

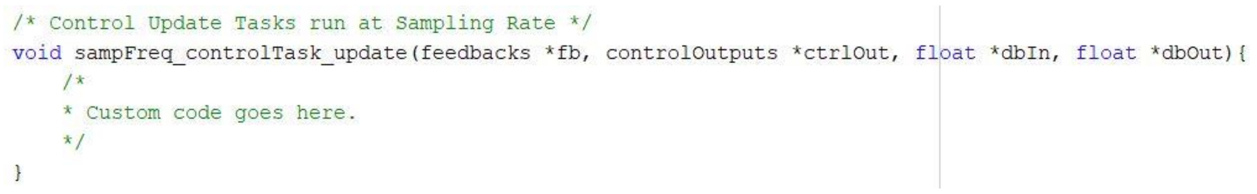
```
/********************** Debug interfaces ******************
*
*                  Do not remove macros; only edit values.
*
**********************************************************/
#define NUMBER_OF_DEBUG_INPUTS          5
#define NUMBER_OF_DEBUG_OUTPUTS         10
```

The default values are set as 5 and 10 for debug inputs and outputs, respectively. Next, open the blockName_interface.c file (unifiGfmController_interface.c in this example). The task groups are defined here. The *Initialization Task* should be defined here-



```
/* Initialization Tasks */
void initializationTasks(void){
    /*
    * Custom code goes here.
    */
}
```

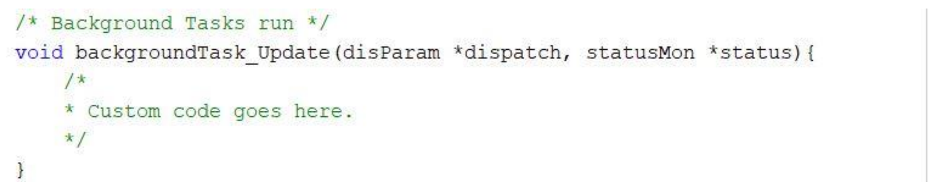
The ***Control Task*** should be defined here-



```
/* Control Update Tasks run at Sampling Rate */
void sampFreq_controlTask_update(feedbacks *fb, controlOutputs *ctrlOut, float *dbIn, float *dbOut){
    /*
    * Custom code goes here.
    */
}
```

Note that the pointers to the I/O structures *Feedbacks, control outputs, debug inputs,* and *debug outputs* are available to the *Control Task.* You may use any or all of them in your code.

The ***Background Task*** should be defined here -



```
/* Background Tasks run */
void backgroundTask_Update(disParam *dispatch, statusMon *status){
    /*
    * Custom code goes here.
    */
}
```

The I/O structures *dispatch parameters* and *status monitors* are available here. For any of the three task groups, you may write your code directly in the blockName_interface.c file or you may link source files from an external library/ code-base. To link external code, e.g., control software in a Texas Instrument's Code Composer Studio (CCS) project, open the

blockName_LinkCustomCode.m (unifiGfmController_LinkCustomCode.m in this example). Add *include paths* as
coder.updateBuildInfo('addIncludePaths','C:\...\...\...');

Add *source paths* as
coder.updateBuildInfo('addSourcePaths','C:\...\...\....');

Add source files, e.g., *dummySourceFile.c,* as
coder.updateBuildInfo('addSourceFiles','dummySourceFile.c');

You may also modify the *Background Task* period; the default value is 5, i.e., $f_B = f_s/5$.

Example: The UNIFI GFM control C-library is taken as an example here to demonstrate the linking of an external code base with the SIL block. Download and unzip the unifiGfmControlLibrary.zip into a folder of your choice. In the blockName_LinkCustomCode.m file, add the include paths, source paths, and source files as

```
coder.updateBuildInfo('addIncludePaths','...\unifiGfmControlCLib\include');
coder.updateBuildInfo('addSourcePaths', '...\unifiGfmControlCLib\source');
coder.updateBuildInfo('addSourceFiles','compensators.c');
coder.updateBuildInfo('addSourceFiles','primaryControl.c');
coder.updateBuildInfo('addSourceFiles','transformations.c');
coder.updateBuildInfo('addSourceFiles','vectorControl.c');
```

For demo model unifiGfmController_LinkCustomCode.m, modify *include path* and *source path* (first two lines) to where *unifiGfmControlLibrary* is installed on computer.

Next, in the blockName_interface.c file define the task groups as

```
/* Initialization Tasks */
void initializationTasks(void){
    stop_vectorControl((unsigned char) 1);
}

/* Control Update Tasks run at Sampling Rate */
void sampFreq_controlTask_update(feedbacks *fb, controlOutputs
*ctrlOut, float *dbIn, float *dbOut){
    update_vectorControl(fb, ctrlOut, dbIn, dbOut);
}

/* Background Tasks run */
void backgroundTask_Update(disParam *dispatch, statusMon *status){
    update_dispatchMonitor(dispatch, status);
}
```

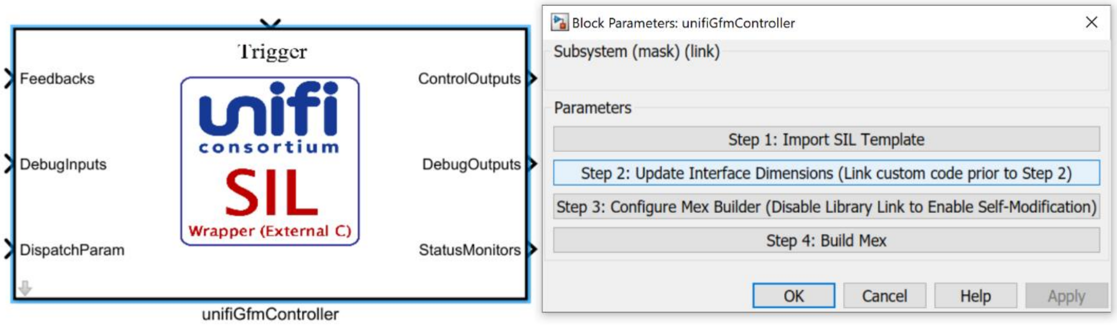
Include the header file containing the function prototypes of the added functions, such as `stop_vectorControl()`, `update_vectorControl()`, and `update_dispatchMonitor()`, as

```
#include "vectorControl.h"
```

Note that no modification in the I/O structures are needed since the SIL template is built for the UNIFI GFM control C library.

Remark: If your custom/external C-code uses fixed-point arithmetic, the type conversions between the fixed-point data and floating-pint data (used by the I/O structures) should be done in the blockName_interface.c file.

vi. Once the custom code/ external code has been linked, update the interface (I/O) dimensions. This step configures the SIL block based on the modifications made in the preceding step.



Trigger
Feedbacks
DebugInputs
DispatchParam
ControlOutputs
DebugOutputs
StatusMonitors
unifi consortium
SIL
Wrapper (External C)
unifiGfmController
Block Parameters: unifiGfmController
Subsystem (mask) (link)
Parameters
Step 1: Import SIL Template
Step 2: Update Interface Dimensions (Link custom code prior to Step 2)
Step 3: Configure Mex Builder (Disable Library Link to Enable Self-Modification)
Step 4: Build Mex
OK
Cancel
Help
Apply

vii. Next, press *Step 3 : Build Mex Builder* and *Step 4: Build Mex* consecutively to generate the mex file in the working directory. Once Step 4 is complete, the SIL block is ready for simulation. Note that the mex file is named after the SIL block name (unifiGfmController_mex.mex64) and it must be in the MATLAB working directory when the simulation is run.

viii. Every time any code changes are made in the custom code/external code, only repeat Step *3 : Build Mex Builder* and *Step 4: Build Mex.*

ix. If the I/O structures are modified, then repeat *Step 2: Update Interface Dimensions* → *Step 3: Build Mex Builder* → *Step 4: Build Mex.*

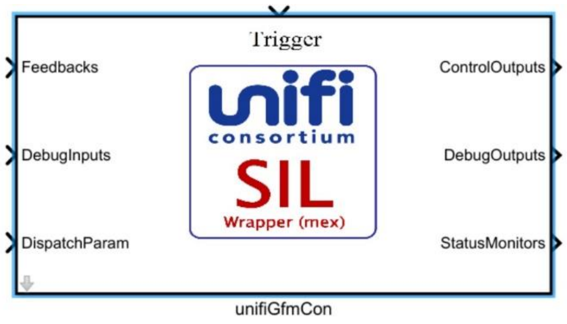
3. Wrapper Block - *UNIFI SIL Wrapper using Mex*

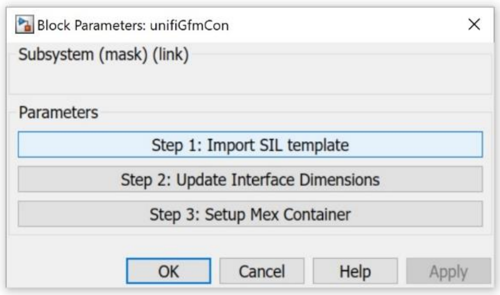
The *UNIFI SIL Wrapper using Mex* block is to be used with an already built mex file. This block is useful for integrating external code as black-box software. To generate the mex file, the *UNIFI SIL Wrapper with Mex Builder* has to be used. For running the *UNIFI SIL Wrapper using Mex* block, two files are necessary – (1) the mex file generated using the *UNIFI SIL Wrapper with Mex Builder,* e.g., unifiGfmController_mex.mex64 built in the previous section, and (2) the corresponding *_interface.h file used for building the mex file, e.g., unifiGfmController_interface.h. While sharing black-box software, these two files have to be shared. The configuration steps for the *UNIFI SIL Wrapper using Mex* block is given as follows-

i. Drag and drop the library block *UNIFI SIL Wrapper using Mex* into your SIMULINK model. Save the model as *.slx file in a folder/directory of your choice and change the MATLAB working directory to the same folder.

ii. Name the SIL block without any space in the name; avoid special characters; avoid numbers as the first letter. For this example, we choose the name *unifiGfmCon.*

iii. In the *Block Parameters* window (double click on the SIL block), the first tab *Step 1: Import SIL Template* creates a folder named *wrapperBuildDir* in the working directory and imports the necessary scripts. Note that all script names start with the name you chose for the SIL block.




Block Parameters: unifiGfmCon
Subsystem (mask) (link)
Parameters
Step 1: Import SIL template
Step 2: Update Interface Dimensions
Step 3: Setup Mex Container
OK
Cancel
Help
Apply

iv. Copy the contents of the original *_interface.h file, e.g., unifiGfmController_interface.h, into the unifiGfmCon_interface.h file.
v. Complete/click *Step 2: Update Interface Dimensions* and *Step 3: Setup Mex Container*.
vi. Copy the pre-built mex file into the MATLAB working directory and rename it after the blockname you chose, e.g., unifiGfmCon_mex.mex64. The SIL block is ready for simulation.

4. Clearing Static Data/Variables

The static/persistent variables/states used by a mex file, i.e., the software being wrapped, are cleared when the SIMULINK model is closed. If your control software does not explicitly initialize/clear static data at the beginning of simulation, i.e., in the *Initialization Task,* to ensure proper initialization each time the simulation is run, static data can be cleared when simulation is stopped. To clear static data at simulation-stop, open the *Stop Function* callback of the SIMULINK model as: Right-click anywhere empty in your model → Model Properties → Callbacks → StopFcn. Add the following script –

```
clear functions
```

5. Additional Configurations

Multiple SIL wrapper blocks can be used in the same SIMULINK model. The *wrapperBuildDir* folder is shared by all SIL blocks in the model. The same function is shared by all SIL blocks to update the interface dimensions. Therefore, make sure that Step 2, Step 3, and Step 4 are completed sequentially each time any change is made in the I/O structures of a *UNIFI SIL Wrapper with Mex Builder* block. For only code changes, only Step 4 is sufficient. Note that if Step 1 is repeated, the mex builder scripts are overwritten; in such an event all following steps including the linking to custom/external code base have to be repeated.