|  |
| --- |
| Swedspot |
| FDT |
| Tutorial |

|  |
| --- |
| v.1.1.0 |

Contents

[1 System requirements 1](#_Toc461708392)

[1.1 Minimum System Requirements 1](#_Toc461708393)

[1.2 Recommended System Requirements 1](#_Toc461708394)

[2 Prerequisites 1](#_Toc461708395)

[3 Creating the First Project 2](#_Toc461708396)

[3.1 Installing FDT 2](#_Toc461708397)

[3.2 Starting FDT for the first time 2](#_Toc461708398)

[3.3 Creating a new project 2](#_Toc461708399)

[3.4 Setting up FDTs environment 3](#_Toc461708400)

[3.5 The FDT Tool 6](#_Toc461708401)

[3.6 Designing the new project 6](#_Toc461708402)

[3.7 Adding Function Blocks to the Project 21](#_Toc461708403)

[3.8 Adding an Icon 24](#_Toc461708404)

[3.9 Adding an Image 25](#_Toc461708405)

[3.10 Adding tell-tales 25](#_Toc461708406)

[3.11 Adding Check Messages to the Project 26](#_Toc461708407)

[3.12 Save the Project 28](#_Toc461708408)

[4 The Simulator 29](#_Toc461708409)

[4.1 Input Controls 30](#_Toc461708410)

[4.2 CAN Input 30](#_Toc461708411)

[4.3 Configure the Simulator 30](#_Toc461708412)

[4.4 Persistent Memory (Simulated EEPROM) 31](#_Toc461708413)

[5 Function Blocks 31](#_Toc461708414)

[5.1 BrakePressureLowTellTale 31](#_Toc461708415)

[5.2 ChargingTellTale 32](#_Toc461708416)

[5.3 EngineOilPressureTellTale 32](#_Toc461708417)

[5.4 FuelConsuption 32](#_Toc461708418)

[5.5 FuelLevel 33](#_Toc461708419)

[5.6 HighBeamTellTale 34](#_Toc461708420)

[5.7 IlluminationManager 34](#_Toc461708421)

[5.8 LocalizationManager 35](#_Toc461708422)

[5.9 OdoAndTripDisplayHandler 35](#_Toc461708423)

[5.10 OdoRuntimeHandler 36](#_Toc461708424)

[5.11 ODOTrip 36](#_Toc461708425)

[5.12 OuptutShaftRotationSpeed 37](#_Toc461708426)

[5.13 ParkBrakeIndicator 37](#_Toc461708427)

[5.14 QuadSwitchManager 38](#_Toc461708428)

[5.15 TPMS 38](#_Toc461708429)

[5.16 TurnIndication 39](#_Toc461708430)

[5.17 UreaLevelMeter 40](#_Toc461708431)

[5.18 VehSpdGauge 40](#_Toc461708432)

[5.19 WarningIconFields 41](#_Toc461708433)

[5.20 WaterTempHighIndicator 41](#_Toc461708434)

[6 Creating a Function Block 42](#_Toc461708435)

[6.1 Basic Parameters 42](#_Toc461708436)

[6.2 Configure Input Signals 42](#_Toc461708437)

[6.3 Configure Output Signals 42](#_Toc461708438)

[6.4 Setup Configuration Parameters (design-time) 42](#_Toc461708439)

[6.5 Setup Calibration Parameters (run-time) 43](#_Toc461708440)

[6.6 LabelMappings 43](#_Toc461708441)

[6.7 Icon Resources 43](#_Toc461708442)

[6.8 Review Function Definition 43](#_Toc461708443)

[6.9 Adding Tasks/Signals/Parameters after running the wizard 43](#_Toc461708444)

[7 Widgets 46](#_Toc461708445)

[7.1 Label widget 47](#_Toc461708446)

[7.2 Icon widget 47](#_Toc461708447)

[7.3 Image Widget 48](#_Toc461708448)

[7.4 Dynamic Icon widget 49](#_Toc461708449)

[7.5 Blockbar widget 49](#_Toc461708450)

[7.6 Column Level widget 50](#_Toc461708451)

[7.7 Dynamic Text Label widget 51](#_Toc461708452)

[7.8 Dynamic Value widget 51](#_Toc461708453)

[7.9 Dynamic Label widget 52](#_Toc461708454)

[7.10 Dynamic Image Widget 52](#_Toc461708455)

[7.11 Selectable List widget 53](#_Toc461708456)

[7.12 Signal Trigger widget 54](#_Toc461708457)

[7.13 List Navigation Widget 55](#_Toc461708458)

[8 Animations 55](#_Toc461708459)

[8.1 Frame-by-Frame animation 56](#_Toc461708460)

[8.2 Opacity animation 56](#_Toc461708461)

[8.3 Translation animation 56](#_Toc461708462)

# System requirements

## Minimum System Requirements

* Operating system: Microsoft Windows 7
* CPU: Intel Core 2 family (Duo, 2 GHz+) or equivalent (x86)
* Graphics/screen resolution: 1280x1024 (SXGA), any on-board graphics chipset
* RAM: 4 GB
* Disk space: 5 GB for components, 10 GB free on system

## Recommended System Requirements

* Operating system: Microsoft Windows 7
* CPU: Intel Core i3 family (2 cores) or better (x86)
* Graphics/screen resolution: 1920x1080 (full-HD), any on-board graphics chipset
* RAM: 8 GB
* Disk space: 5 GB for components, 10 GB free on system

**Note:** As the FDT tool suite generate/compile source files the tool suite’s performance is tightly bound to the system file access speed. A hard disk of SSD type is therefore highly recommended and it shall also be noted that any software (like for example antivirus programs, and run-time system encryption) can have adverse/unfavorable effects on the tool suite’s performance.

# Prerequisites

In order to use the FDT tool the following software needs to be installed:

* **MinGW** - Windows port of GCC  
  1. Download and install from: <http://sourceforge.net/projects/mingw/files/latest/download?source=files>
  2. When the installation is finished a program called “**MinGW Installation Manager**” will start, mark the “**mingw32-base**” and “**mingw32-gcc-g++**” packages for installation by clicking the checkbox next to them. Then choose “Installation-> Apply Changes” to install the packages.
  3. Add the MinGW bin (default: **C:\MingGW\bin**) path to the Windows path.
* **Vector XL Driver Library** - Driver Library for Vector Hardware Interfaces  
  1. Download and install “**Vector Driver Setup 9.\*.\***” from: <https://vector.com/vi_downloadcenter_en.html?type=Driver&formular_treffer_submit=1>
     + Select “**CANcaseXL**” when prompted for driver selection.
  2. Go to the Windows Control Panel and open “**Vector Hardware Config**”. Create an application called “**BSPSim**” with default settings. The hardware should list a "**Virtual CAN Bus 1**" with two channels, right click "Channel 1", and select BSPSim -> CAN 1.

# Creating the First Project

## Installing FDT

Install the FDT software on the PC by running the installer called FDTInstaller.

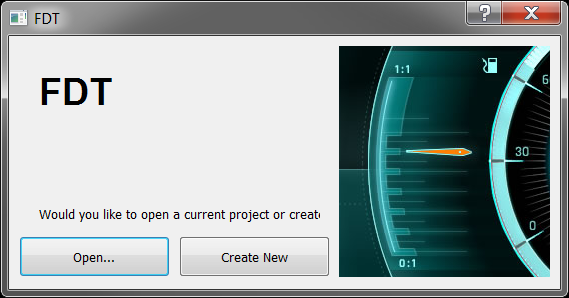
## Starting FDT for the first time

The first time FDT Tool is started the user will be asked to point out a function library path. This path will automatically be stored and saved in the user settings.

The function library contains the function blocks that will be used to build the platform.

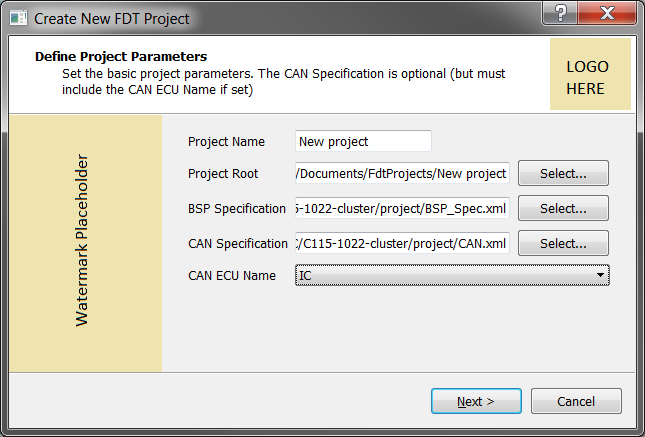
## Creating a new project

After the function library path has been set, or if FDT already have a configured function library path, the user will be prompted to create or open a project.



Click “**Create New**” to start a new project.

The following window will appear:



Choose a project name, root folder and BSP specification (XML file created with included BSP Editor). These are mandatory inputs.

If CAN will be used also select a CAN Specification and choose ECU name “**IC**” in the dropdown that appear after the CAN specification has been selected.

***Note****: Selecting a CAN specification is optional, but if a CAN specification is chosen it is required to set the CAN ECU Name accordingly*.

Click **“Next“,** and review the project settings, if everything looks good click **“Finish”** to open FDT’s main window.

## Setting up FDTs environment

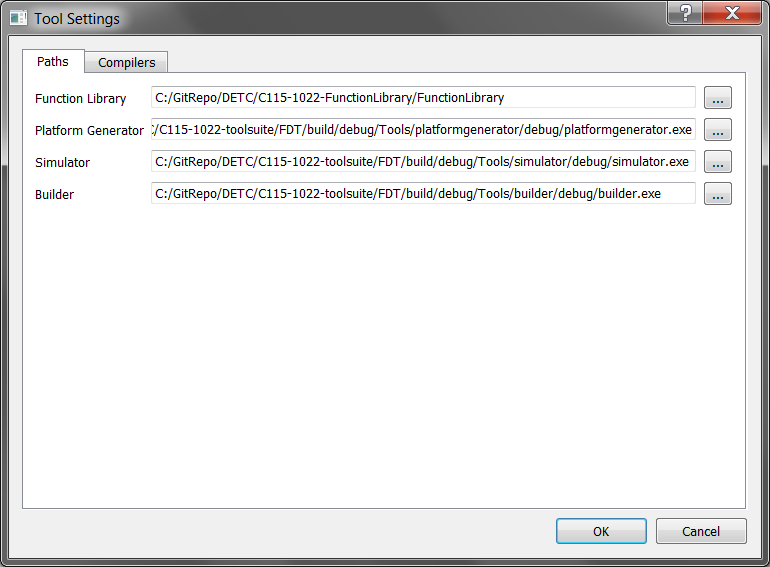
To use the FDT its environment needs to be setup for the current system. Such as pointing out the paths to tools and compilers. It is also required to specify some settings for each project, as well as some user specific settings. This includes project build targets and what compiler to use for each of them.

### Tool Settings

From the main view, open **Tools->Settings**

#### Paths

In the settings dialog that opens there are two tabs, one for setting paths to the tools and function library and one for adding compilers to use when building the project.

**

***NOTE****: A platform generator (platformgenerator.exe), platform builder (builder.exe) and simulator (simulator.exe) is distributed with FDT. These can be used to generate, build and run the project to test it. They are located in the FDT installation folder.*

##### Function Library path

The path to the function library should already be set since this is done the first time the FDT Tool is started. Otherwise it needs to be specified here.

##### Platform Generator path

The path to the code generator used to generate code for the target platform.

Set platform generator path to: **<FDT-Installation-path>/platformgenerator.exe**

##### Platform Builder path

To be able to build the target code from the FDT Tool a path to the builder is needed to be provided here.

Set builder path to: **<FDT-Installation-path>/builder.exe**

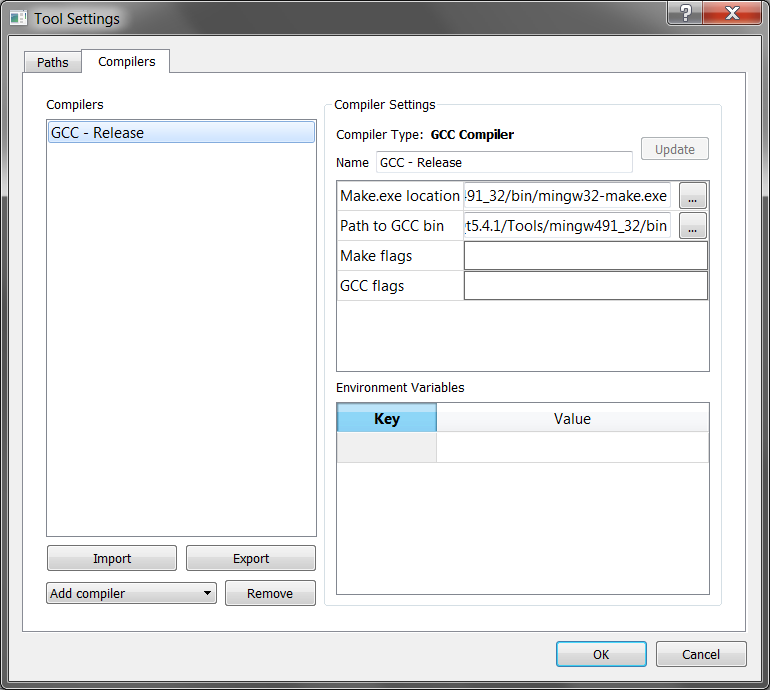
##### Simulator path

To be able to verify the project in a simulator before applying it to target a path to the simulator needs to be added here.

Set simulator path to: **<FDT-Installation-path>/simulator.exe**

#### Compilers

Some compiler settings needs to be provided to be able to build the first project directly from FDT. This is done in the Compilers tab.



##### Adding a Compiler

To add a compiler, simply choose a compiler type from the combo box. Then browse the path for the compiler in the compiler settings. Write a name and click save.

***NOTE****: Some types of compilers need additional configurations, such as location of its bin folder. Optional settings can also be provided, i.e. compiler/maker flags and environment variables.*

Select “**GCC Compiler”** in the combo box and configure as follows:

* Name: **GCC**
* Make.exe location: **<MinGW-installation-path>/bin/mingw32-make.exe**
* Path to GCC bin: **<MinGW-installation-path>/bin**

Click “**Save”** and then “**OK”** to close the tools settings window.

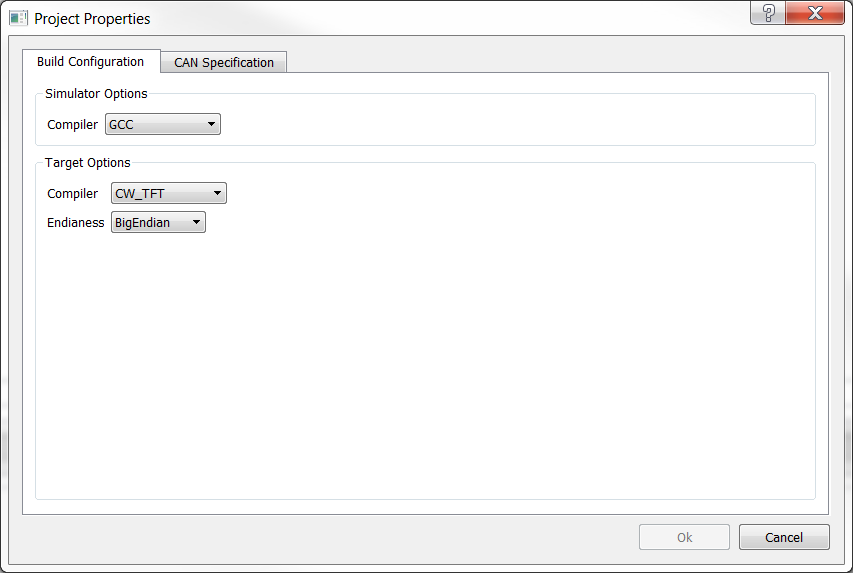
### Project Properties

From the main view, open **Project->Properties**

In the project properties dialog that opens there are two tabs, one for configuring simulator and target, and one for changing the CAN specification.

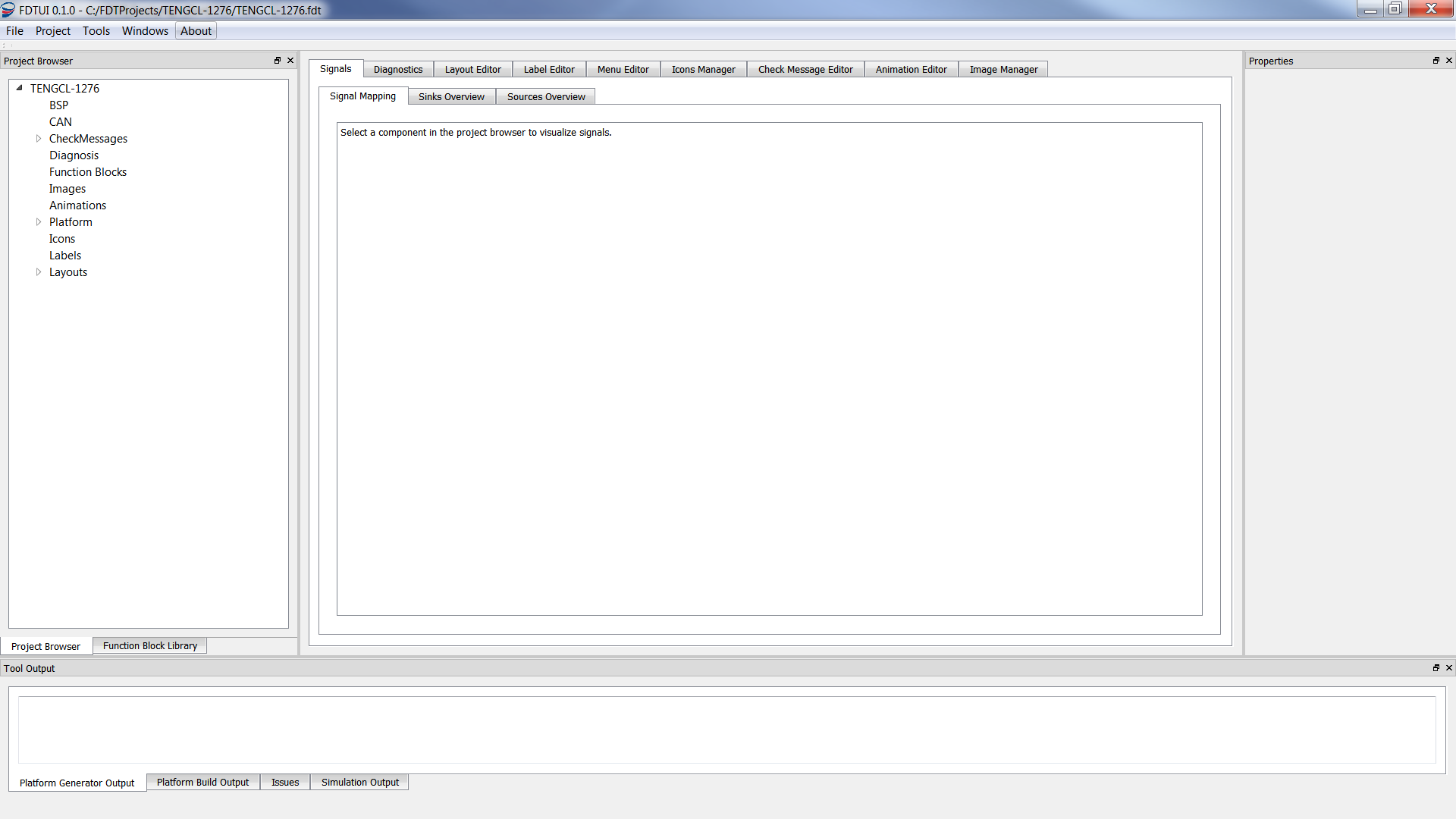
#### Build Configuration

This is where the compiler for the simulator and target is specified.



For the target it is also needed to specify the endianness of the hardware.

## The FDT Tool



As default the main window consists of 4 areas:

* Left area: To the left there is an area that consists of two tabs: “**Project Browser**” and “**Function Block Library**”.
* Middle Area: In the middle is where all the editors are located.
* Right Area: To the right is as a “**Properties Window**” that will show the properties for the item selected in the left or middle area.
* Bottom area: At the bottom there are a number of different output windows that displays output from the code generator, compiler, simulator, etc.

The different areas and editors will be described later in the document.

It is possible to move the Properties window and have it as a separate window or to add it to the Left area as a tab together with the Project Browser and the Function Block Library tab.

## Designing the new project

### Using displays

To be able to use a display, the display needs to be described in the BSP specification. There is support for three types of displays:

* Monochrome Display (Software Rendered)
* TFT Display (Hardware Rendered)
* Segments Display

#### Configure Display Settings

##### Monochrome Display

To configure the display click on “**Layouts**” in the Project Browser. This will display the available display settings in the Properties Window.

***Note:*** *Not all display properties can be changed. The uneditable properties are the properties specified in the BSP specification selected at project creation.*

The first section contains properties read from the BSP specification and these cannot be updated in the tool:

* **Rendering Type** SW
* **Height**
* **Width**
* **Bits per Pixel** *How many bits used to represent one pixel*
* **Use DCU Memory** false
* **Rotate Screen 90 degrees** false

**“Rotate Screen 90 degrees”** is set in the BSP specification and is used for compensating for screens that do not have the origin set to the upper left corner.

After that follows the properties that are editable by the developer. Configure the display with the following settings:

* **CropLeft**: 0
* **CropRight**: 0
* **CropTop**: 0
* **CropBottom**: 0
* **Main Layout** *The Layout shown after startup*
* **Startup screen enabled** Unchecked
  + **Startup screen timeout** *How many seconds the startup layout should be shown*
* **SystemPowerMode** -> Platform.PowerMode

##### TFT Display

To configure the display click on “**Layouts**” in the Project Browser. This will display the available display settings in the Properties Window.

***Note:*** *Not all display properties can be changed. The uneditable properties are the properties specified in the BSP specification selected at project creation.*

The first section contains properties read from the BSP specification and these cannot be updated in the tool:

* **Rendering Type** HW
* **Height**
* **Width**
* **Bits per Pixel** *How many bits used to represent one pixel*
* **Use DCU Memory** true
  + **DCU Memory Size**
  + **DCU Number of Layers**
* **Rotate Screen 90 degrees** true

After that follows the properties that are editable by the developer. They are the same as for the Monochrome Display:

* **CropLeft**:
* **CropRight**:
* **CropTop**:
* **CropBottom**:
* **Main Layout** – The Layout shown after startup
* **Startup screen enabled** – Defines if the startup screen should be shown on startup or not.
  + **Startup screen timeout** - How many seconds the startup layout should be shown
* **Chroma key** – Defines a range of colors that will be set to transparent for all images and also widget that have the “Transparent” property set.
  + **Red Range** – The Chroma key Red Range
  + **Green Range** – The Chroma key Green Range
  + **Blue Range** – The Chroma key Blue Range
* **Input Signals**
  + **SystemPowerMode** -> Platform.PowerMode
  + **CurrentLanguage** -> LocalizationManager.CurrentLanguage
  + **Up** – QuadSwitchManager.Button1Event
  + **Down** – QuadSwitchManager.Button2Event
  + **Left** –
  + **Right** –
  + **Set** – QuadSwitchManager.Button3Event
  + **Clear** – QuadSwitchManager.Button4Event

###### Layers

For hardware rendered platforms FDT is designed to use layers to create the layouts. The number of layers supported by a platform is found in the specification for the DCU used by the platform.

The FDT tool allocates layers based on the areas that are setup for each layout in the project. If an area contains widgets that are software rendered then the area will allocate a layer.

Some widgets require their own layer so an area will then use more than one layer.

On the other hand, if a layer only contains widgets that requires their own layer, then the area will not allocate any layer of its own, only the widgets inside the area will allocate layers. An area that doesn’t allocate a layer doesn’t support having a border around it. It will cause a validation failure when trying to generate such project.

It’s important to take into account the number of available layers when designing a layout. If the layout requires more layers than available in the DCU then the FDT tool won’t be able to build the project and it will also generate a validation error.

The following widget will allocate one or more layers on their own:

* *Image widget*
* *Dynamic Image Widget*

##### Segment Display

To configure the display click on “**Segment Display**” in the Project Browser. This will display the available settings in the Display Properties window.

***Note****: If no segment display exists in the BSP specification “Segment Display” will not be shown in the Project Browser.*

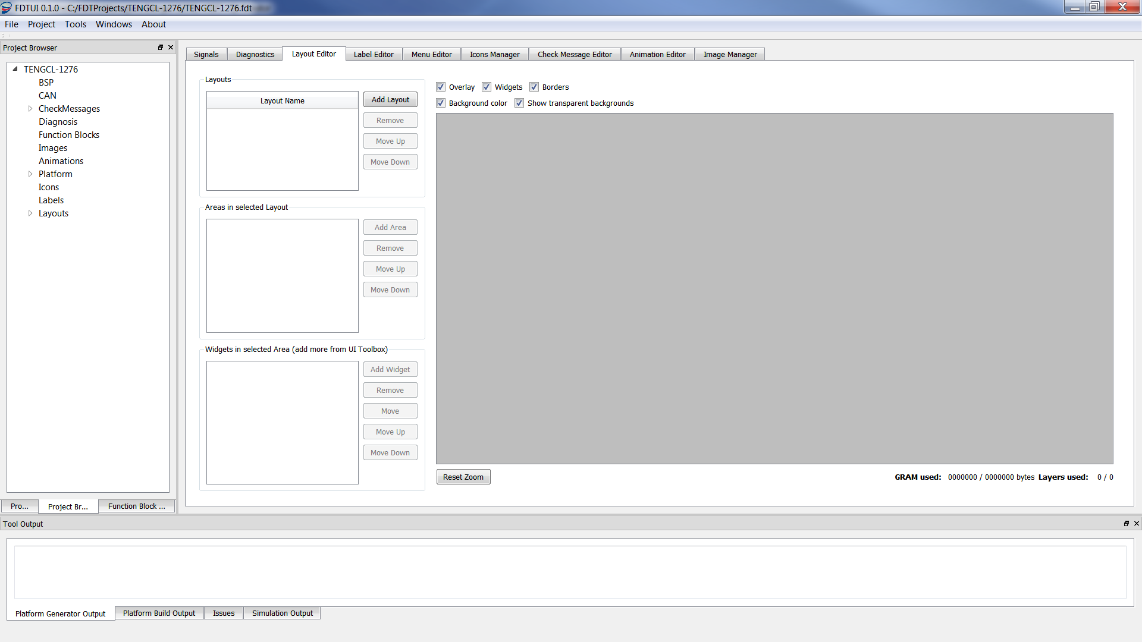
If the segment display is to be used, configure the display with the following settings (or what is required for the current project):

* **PowerMode** -> Platform.PowerMode
* **DisplayValue** -> OdoRuntimeHandler.Value
* **DOTSegmentStatus** -> OdoRuntimeHandler.ShowDot
* **T1SegmentStatus** -> OdoRuntimeHandler.ShowHourGlass

Note that functions blocks are needed before being able to connect the input signals. This will be described later in this document

#### Create Layouts

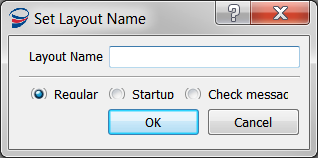
Now it’s time to set up the layout of the display. Click on the tab “**Layout Editor**”.



The tool supports many different layouts. These are configured by selecting Layout Editor tab or by selecting “**Layouts**” in the “**Project Browser**”.

##### Add new Layout

To create a new Layout click on the Add button next to the Layout control box. It will open the following dialog box:



There are three types of Layouts:

* **Regular** – A regular layout used in most cases
* **Startup** – The Startup Layout that will be shown during startup. The timeout for the startup layout is defined in the properties window for Layouts found in the Project Browser.
* **Check Message** – A check message layout that can be used to display important info to the driver based on signals. Check Messages will be explain later in chapter [Adding Check Messages to the Project](#_Adding_Check_Messages).

Add a regular layout called “**Main Layout”**.

Click on one of the newly created layouts and a white square will appear in the grey area. This is the preview area that will show the design of the current layout. Also, the Properties window will be filled with the properties for the Layout:

* **Name** – The name of the Layout.
* **GCL Name** – The GCL component name of the Layout.
* **Background Color** – Only shown for the TFT display. Defines the background color of the layout in 24-bit RGB.

For regular Layouts there are some more properties:

* **Button Transitions** – Is used to set up navigation between layouts using buttons. [Button Transitions](#_Button_Transitions)
* **Signal Transitions** – Can be used to activate a layout when a certain condition is met. [Signal Transitions](#_Signal_Transitions)
* **Output Signals** – Can be used to trigger events when buttons are clicked.

When using transitions it is very important to take the priority of the different layouts into account. For more information about this read chapter [Layout Priorities](#_Layout_Priorities).

##### Add Area

An area is a container for a number of design widgets. It’s a way of creating groups of widgets that are related to each other. All widgets needs an area to be placed in.

Add a new area by clicking on the Add button next to the Areas control box.

The new area will get the default name “**New Area**“, and the size will be set to the size of the display. To change the properties for an area, click on it in the Area box and the properties will appear in the Properties Window.

Also note that the new area is shown in the preview window.

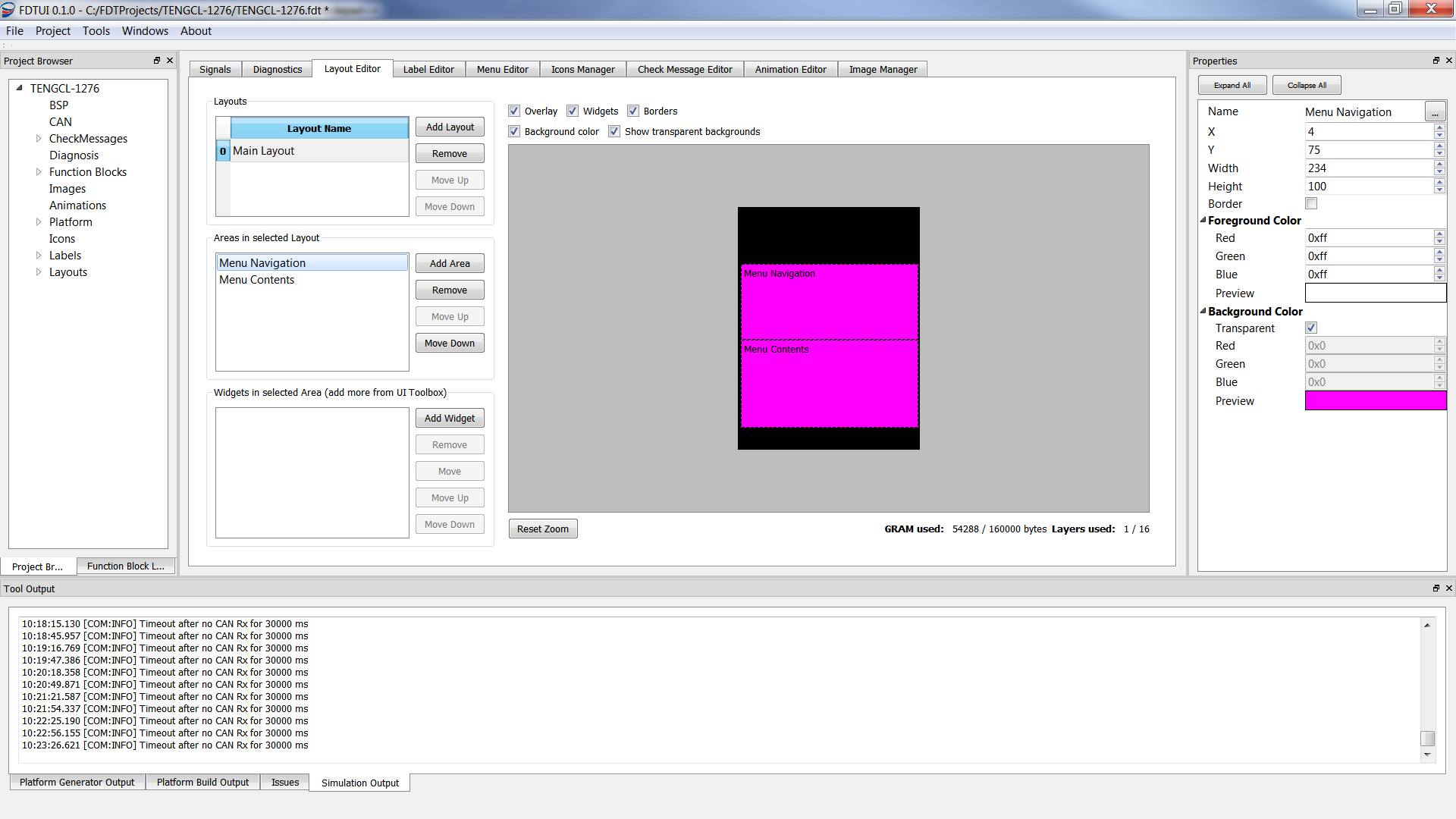
Rename the area to “Date” andconfigure it with the following settings:

* **X**: 4
* **Y**: 3
* **Width**: 135
* **Height**: 27
* **Border**: checked
* **Foreground Color** The color of the border if used
* **Background Color** Set to Transparent by default

Create and configure the following areas:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Name | X | Y | Width | Height | Border |
| Menu Navigation | 4 | 75 | 234 | 100 | No |
| Menu Contents | 4 | 175 | 234 | 115 | No |

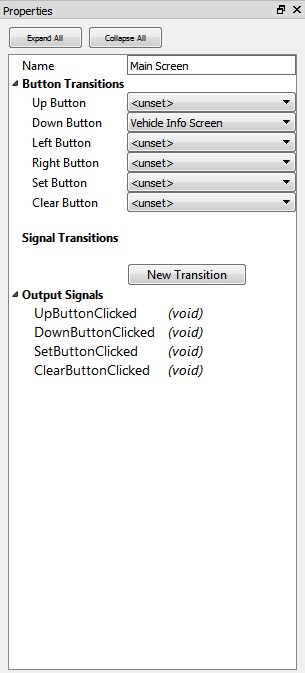
If everything was added correctly the layout editor should now look like this:



#### Adding Layout Transitions

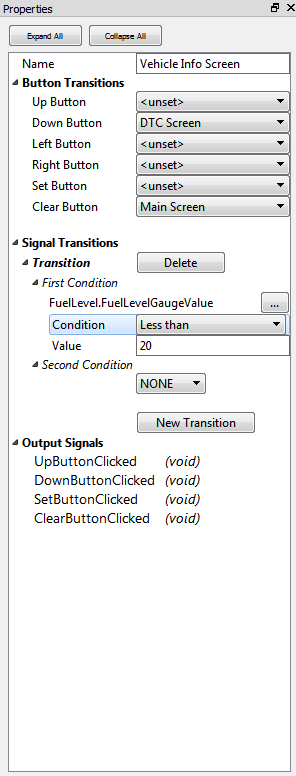
The Layout Transition feature is what makes multiple layouts navigable. The Layout Transition feature supports button navigation between layouts and layout signal transitions based on developer-configured signal conditions.

##### Button Transitions



To configure a button transition, select a layout from the layout overview list in the “**Layout Editor**”. The selected layout will have its button transition properties shown in the associated property view. Assigning a button to a layout has the effect of creating a transition *from* the layout associated with the property view. The transition will persist after the button has been released.

##### Signal Transitions



Transitions can also be triggered by signals. These transitions can be configured in the **“Layout Properties Widget”**, where it is possible to create basic logical clauses to trigger transitions. Note that the signal transitions specifies when the selected layout becomes active, in contrast to the button transitions, which specifies transitions *from* the selected layout.

Signal transitions are not persistent. Should signal input change in such a way that a previous logical expression no longer evaluates to true, the **“Layout Manager”** may select a layout to be shown. If applicable, a layout previously selected as “active” through a button transition, or a higher prioritized layout by the layout list ordering will be shown. Consult section 3.6.1.3.3 on Layout Priorities for more information.

###### The logical framework of Signal Transitions

In the event that multiple layouts have transition conditions which evaluate to true, the uppermost layout in the **“Layout Editor”** layout selection box has priority over other layouts.

Each transition clause has one or two conditions joined by a logical connective.

Conditions consist of three parameters:

* **Signal**: A signal to perform the comparison on
* **Condition**: The comparison operator
* **Value**: The value to test against the signal using the comparison operator

A transition clause is formed by selecting one of the three supported logical connectives.

|  |  |
| --- | --- |
| **Logical connective** | **Logical evaluation** |
| NONE | *P1* |
| AND | *p1* \wedge *p2* |
| XOR | *p1*⊕ *p2* |

The tool supports any number of transition clauses automatically joined together by the OR operator.

Consider using function blocks to define more advanced transition conditions. It is possible to trigger transitions based on the output of user-defined function blocks.

##### Layout Priorities

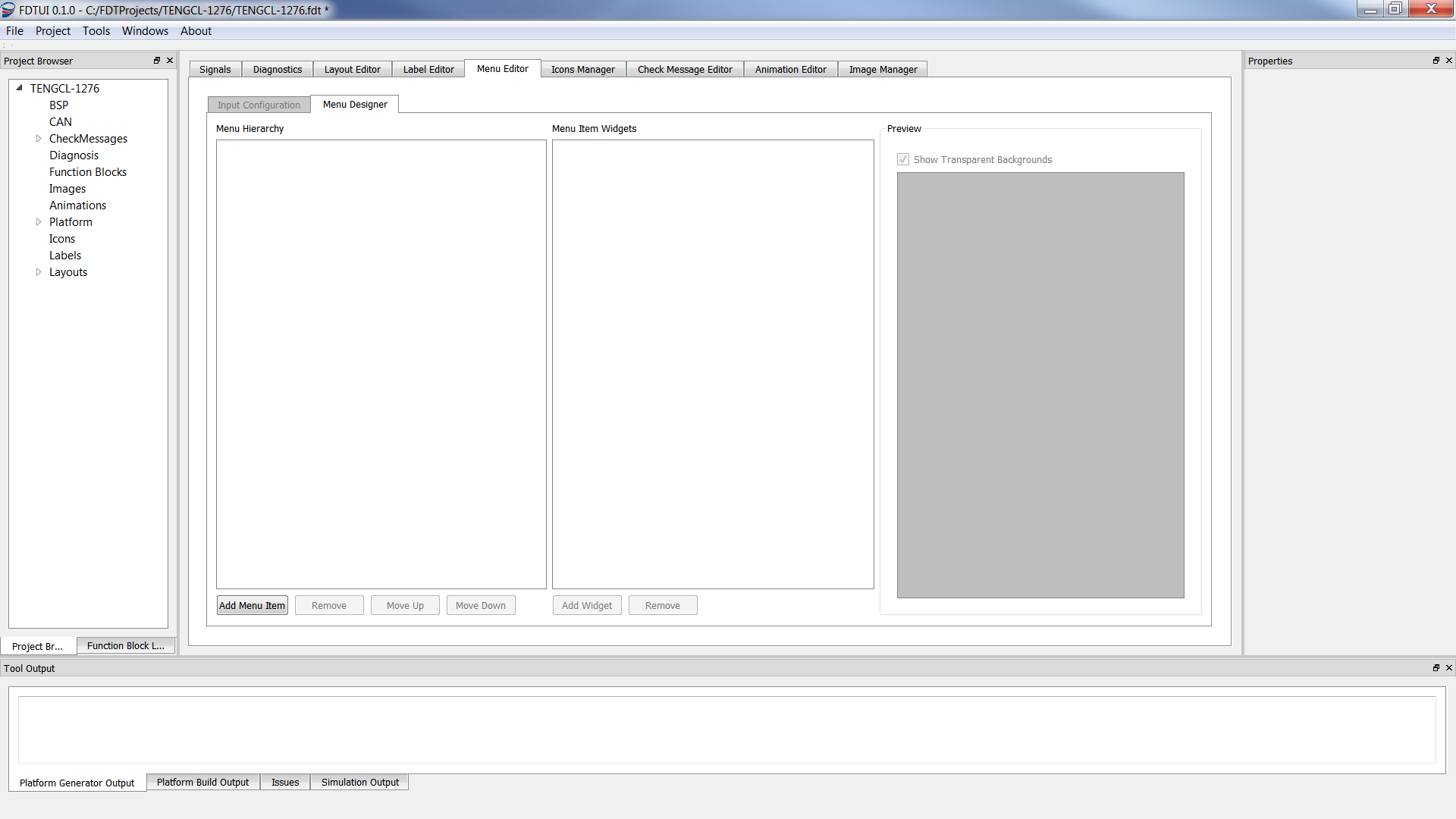
Projects with multiple layouts, signal transitions and button transitions may lead to states where multiple layouts are configured to be active concurrently. In these cases, the system Layout Manager selects one layout to be active according to the algorithm described below:



The “**Layout Manager**” is a system task driven by the system scheduler. The layout selection scheme is therefore run repeatedly and continuously.

#### Adding a Menu

Open the Menu Editor tab.



To add a menu click the “**Add Menu Item**” button. This will open a dialog box where the developer gets to specify the area in which the menu contents should be displayed. Choose the desired area, for example “**Menu Content**” created earlier in this document, and click ok.

In the “**Menu Hierarchy**” box there should now have appeared a “**Menu Root**” object. This is the container for the menu.

***Note****: The tool only supports one menu per project.*

#### Configuring menu input

Navigate to the sub tab “**Input Configuration**” and configure the input signals (using the combo boxes) as follows:

**Up**: Previous Menu Item  
**Down**: Next Menu Item  
**Left**: Unmapped  
**Right**: Unmapped  
**Set**: Enter Submenu  
**Clear**: Leave Submenu

For the menu navigation to work it also needs to have signals connected from a function block. See the chapter [Connect Signals](#_Connect_Signals) for more information about which signals to connect for the menu.

#### Adding the Menu Navigation widget

To be able to navigate the menu there also needs to be a navigation widget. To add a navigation widget select the “**Menu Navigation”** area in the Layout Editor and add the desired type of navigation widgets.

##### List Navigation Widget

Configure the List Navigation widget with the following properties:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Name** | **X** | **Y** | **Width** | **Height** | **Boxed** | **Display Item Count** |
| MenuNavigation | 1 | 1 | 233 | 93 | No | 4 |

***NOTE****: In order for the navigation widget to display properly, a label has to be assigned to each menu item.*

##### Navigation using Dynamic Widgets

###### Dynamic Text Label Widgets

It is possible to use Dynamic Text Label widgets for presenting the menu navigation in an area. The Menu Controller supplies 5 sources for displaying the current menu time, the two next menu items, and the two previous menu items on the same level in the menu hierarchy.

***NOTE****: In order for the Dynamic Text Label widget to display properly, a label has to be assigned to each menu item.*

###### Dynamic Image Widgets

For projects that uses hardware rendering it is also possible to use Dynamic Image widgets to present the menu navigation. It is possible to use up to 5 dynamic image widgets and they should be connected to the five sources of the MenuController that supply image IDs.

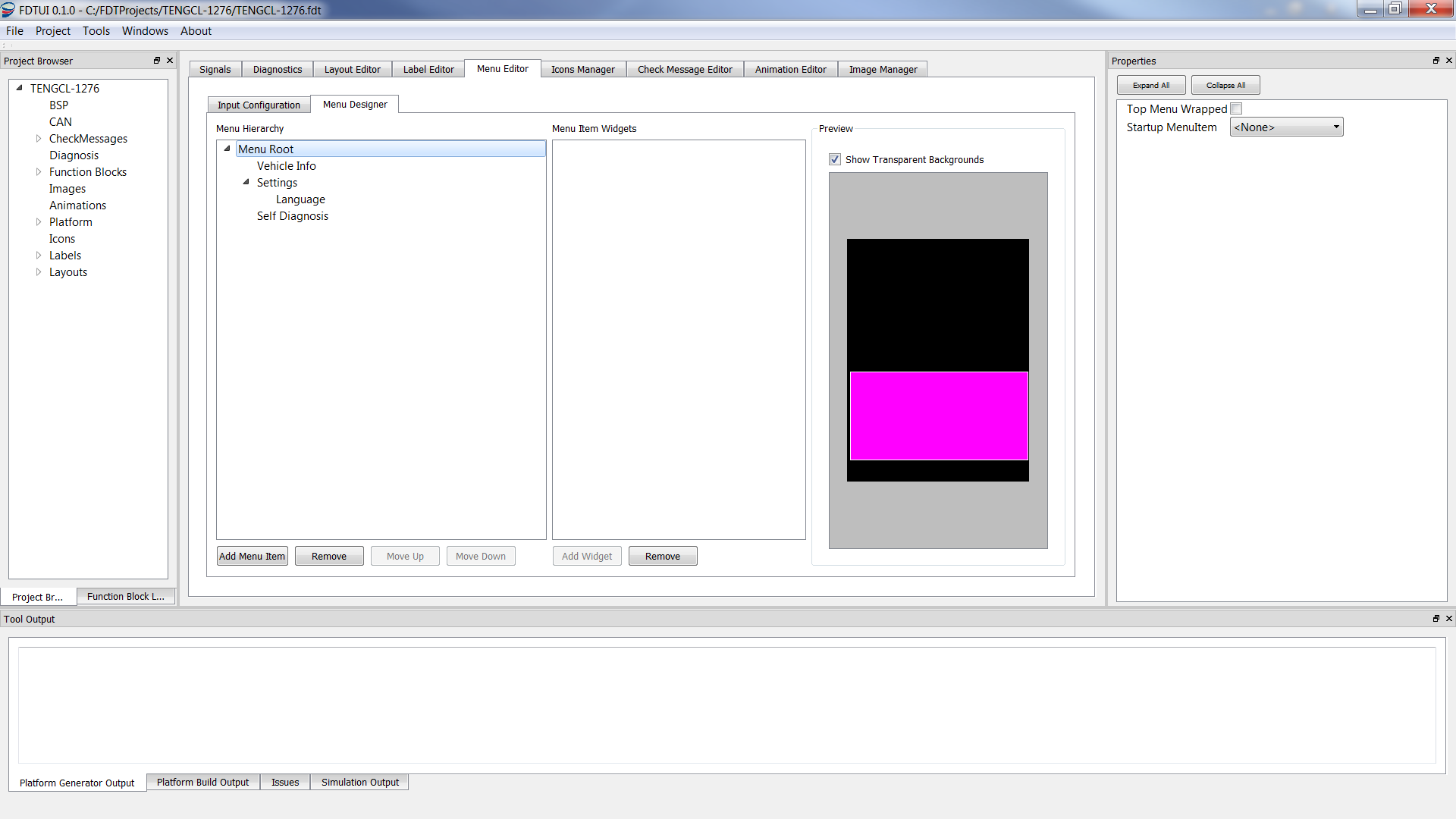
Add the number of dynamic image widgets you want to represent your menu. Connect them to the different sources of the Menu Controller. If for example 3 widgets are used the following menu controller sources should be connected:

* **PreviousItemImage**
* **CurrentItemImage**
* **NextItemImage**

***Note****: This requires that each menu item has an image assigned to it. See chapter* [*Adding an Image*](#_Adding_an_Image)

#### Designing the menu

Navigate to the “**Menu Editor**”tab in the main window. Navigate to the sub tab “**Menu Designer**”.



This is where the menu structure and behavior is defined.

Initially the menu designer only displays a “**Menu Root”** object. This is the container for all menu items in the project. When selected it will provide the following properties in the property window:

* **Top Menu Wrapped** – If top level navigation is enabled, pressing “Next” on the last menu item will make the first menu item selected and pressing “Previous” on the first menu item will make the last menu item selected. If top level navigation is disabled, this setting has no effect.
* **Startup MenuItem** – This determines the menu item which to display when starting the cluster.

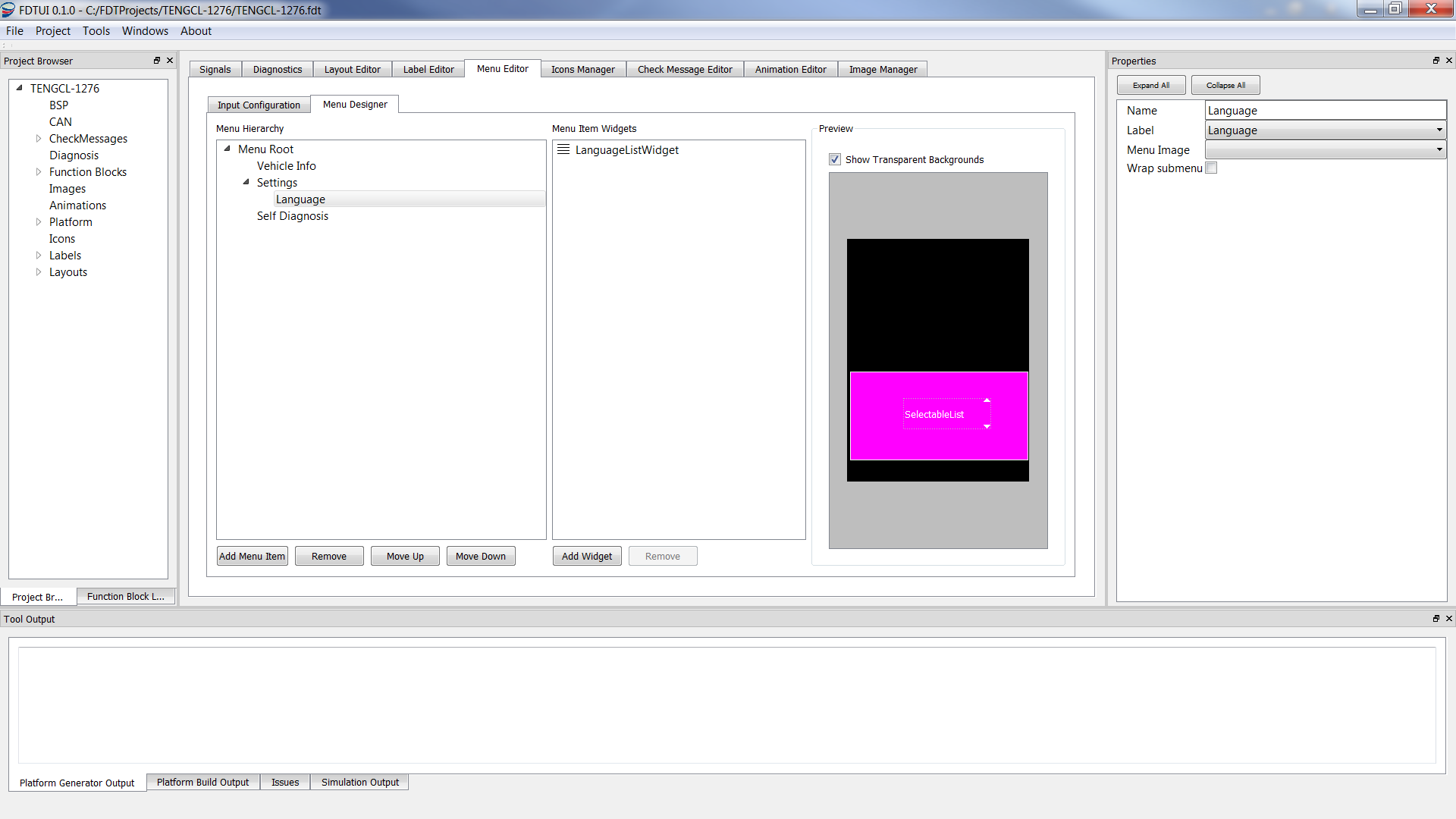
Next step is to add some menu items to the menu. Start by adding the first menu item by clicking the “**Add Menu**”-button in the “**Menu Designer**” tab. The new menu item has the following available properties:

* **Name** – A name for this menu item (**Note:** *The tool doesn’t support spaces in the name!*)
* **Label** – Thelabelto display in the navigation widget or in a Dynamic Text Label widget
* **Menu Image** - The image to display in a Dynamic Image widget
* **Wrap Submenu** – Works in the same way as “**Top Menu Wrapped”** on the Menu Root, but for submenus to this menu item.

When clicking “Add Menu” the new menu item is added as a child to the currently selected menu item. Add a couple of more menu items to create the following menu hierarchy. Also add and configure a **“SelectableListWidget”** as described in chapter [Add contents to the Menu Items](#_Add_contents_to).

* **Menu Root**
  + *VehicleInfo*
  + *Settings*
    - *Language*
      * *SelectableListWidget* ( name: LanguageListWidget, x: 70, y: 35, width: 115, height: 40, selectable: checked)
  + *SelfDiagnosis*

The hierarchy seen in the editor will be the same when running the IC software. The navigation widget will take care of visualizing the hierarchy, while the contents (e.g. widgets) is displayed in the menu contents display area.

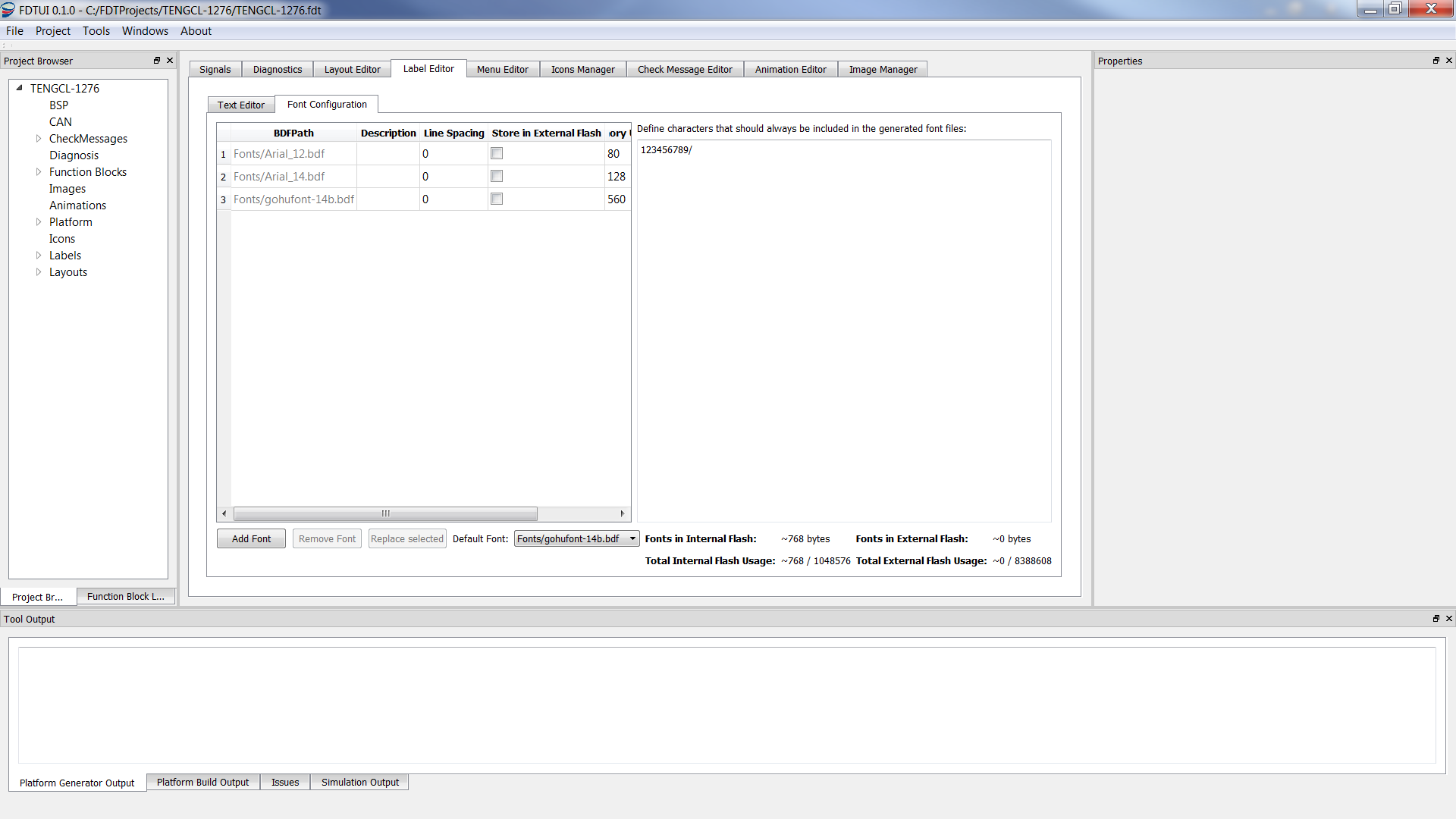


##### Label Editor

In a new project there will be no default labels defined so new labels have to be created in the Label Editor that is found as a tab in the main content area of the FDT tool. The label editor is the main source for all static labels that will be displayed in the cluster.

###### Add Fonts

At least one font has to be added in the “**Font Configuration”** before any labels can be created.



Add a font by specifying the BDF file path. The Description will be the name representing the font in the “**Text Editor**” tab. The font files should be located in the “**Fonts**” folder in the project folder. The “Default” radio button defines which font that should be default when creating a new label, also it works as a default font in UI elements where no font has been defined.

Mandatory Characters

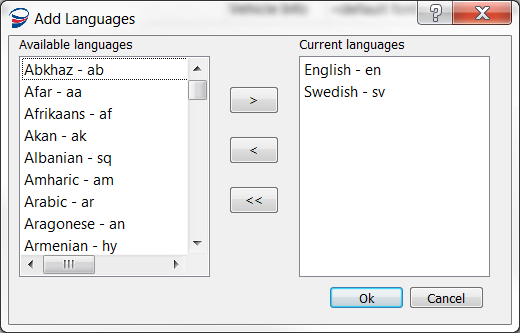
To be able to support Dynamic Text and Dynamic Value labels it is possible to define characters that should always be generated for each font it they exists. This is done in the text box to the right.

Font size in memory

It is possible to store fonts both in the internal and in the external memory. To help the developer with deciding where to store the fonts there is a size approximation done for each font. This is not an exact calculation since the size depends on optimizations made by the compiler at build time but it’s meant to give the developer a hint of the size.

###### Add languages

The next step is to add one or more languages. This is done by pressing the “**Add/Remove Language**” button in the “**Text Editor**” tab.



Add the required languages and click **“OK”** to close the dialog.

###### Add Labels

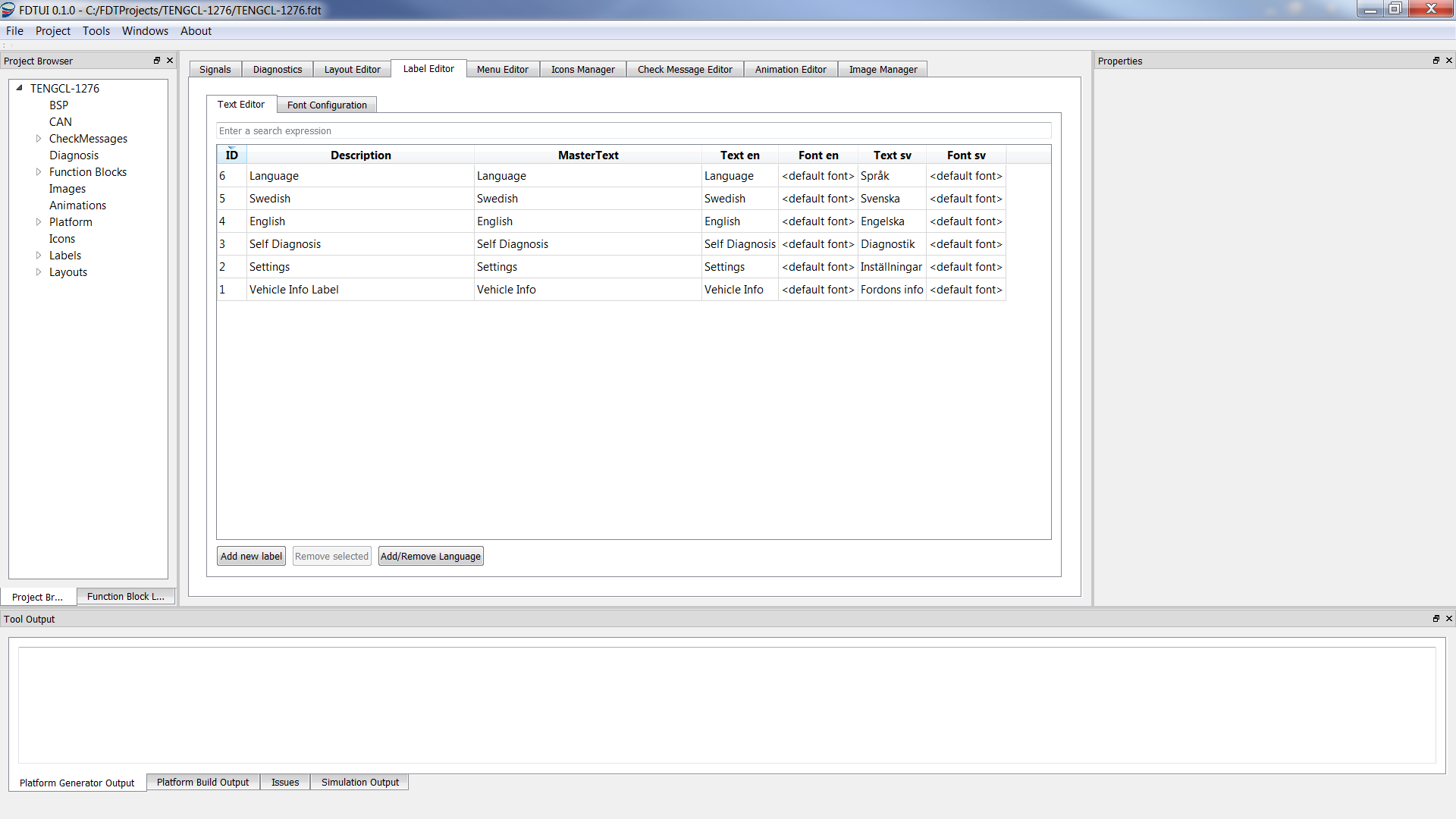
Now it possible to start adding labels. A label has the following attributes:

* **Description** – The name of the label
* **Master Text** – The text that will be displayed if the label doesn’t exist in a certain language

Besides these attributes a label also contains one text field and one font field per language. The text field contains the label written in the specific language and the selected font file needs to support the current language.

Add the following labels in English (en) and their translation to a second language:

* Vehicle Info
* Settings
* Self Diagnostics
* English
* Swedish
* Language

******

***Notice****: If no text has been specified for a language and the project is built, the tool will remove the empty language option.*

#### Add Labels to the Menu Items

With labels defined it’s possible to add labels to the menu items. In the “**Menu Designer”**, click each of the menu items and assign the proper label to them.

#### Add contents to the Menu Items

Add contents to menu items by selecting them and clicking the **“Add Widget”** button. This will open a dialog box displaying the available widgets. Select a widget and click OK and it should be added to the Menu Item Widgets window in the Menu Designer. Clicking the widget will display a list of properties, first the positional and common widget properties:

* **Name** – The name of the widget, for easy identification in the menu designer.
* **GCL Name** – The GCL Component Name
* **X/Y** – The x and y position of the widget relative to the menu contents area
* **Width/Height** – The width/height of the widget. Any contents outside will not be displayed.
* **Boxed** – When selected the widget will have a border around it if it is supported by the widget.

***Note:*** *When a color display is used, and if the display uses “Hardware Rendering” it’s also possible to configure the foreground and background color of each widget. This is done by selecting a widget and specifying the RGB values of a desired color in the widget property view.*

After these generic properties comes widget specific properties that determine the contents of the widget.

Before building the project there are some function blocks needed to be able read hardware signals, from for example the buttons. This will be described next.

## Adding Function Blocks to the Project

All available function blocks are found in the “**Function Block Library**” window. To add a function block, expand it and select the fdf-file and click “Add to project”. When added, the function block can be seen in the Project Browser under Function Blocks. For more information about the different Function Blocks see chapter [Function Blocks](#_Function_Blocks).

In the “**Function Block Properties**”-window it’s possible to see the function blocks properties, e.g. Initial delay, periodicity, input- and output-signals. The “**Initial Delay**” is how long it will take from that the scheduler starts until the function block is initialize. The “**Periodicity**” is how often the function block will be run. They are both in milliseconds.

### Connect Signals

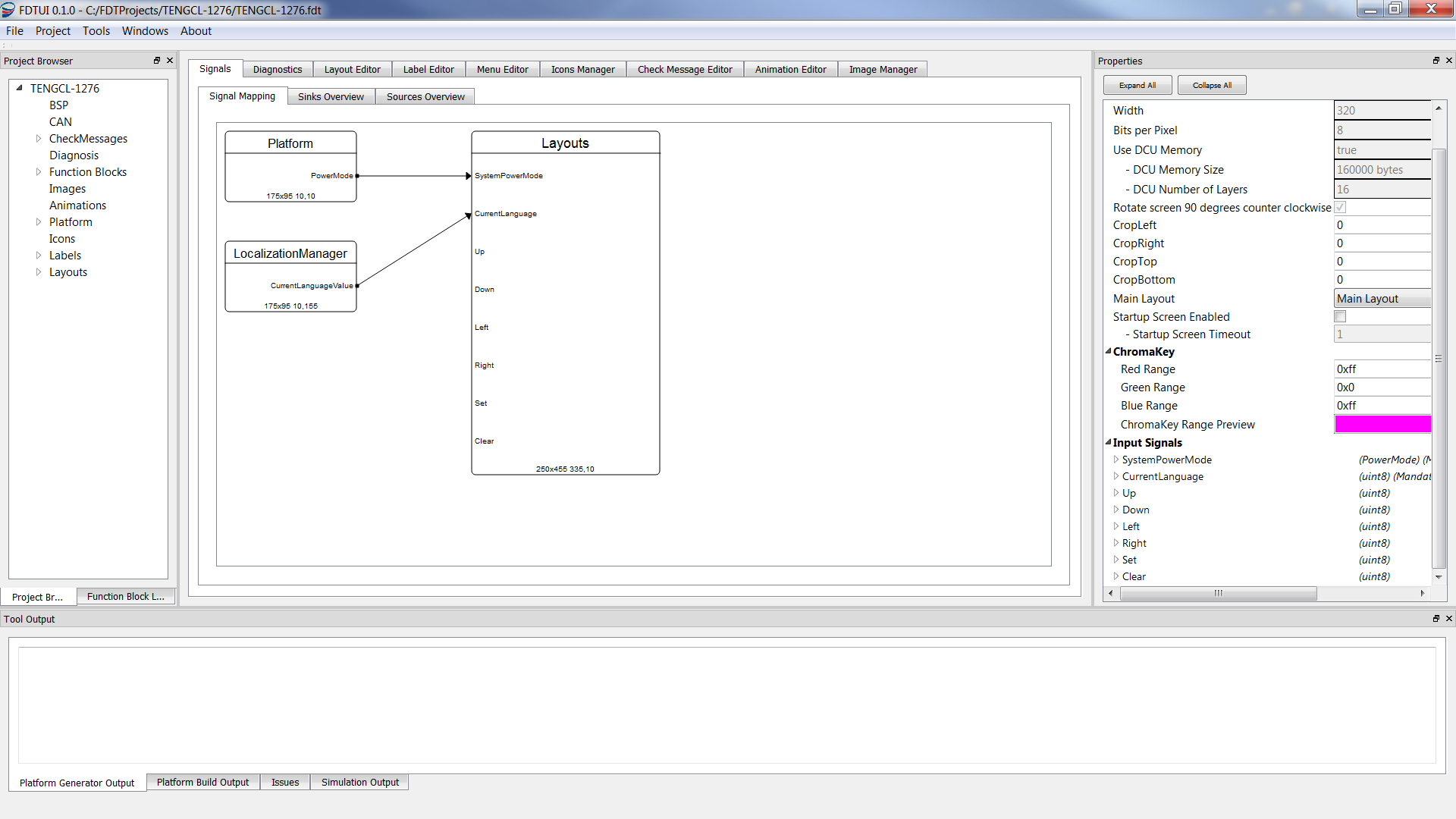
So far there are no signals connected which can be seen in the “Function Block Properties”-window to the right on the screen or in the Signals tab in the middle of the main screen.

#### Signals Window

The “Signals” tab provides tools for connecting and view sinks and sources.

##### Signal Mapping

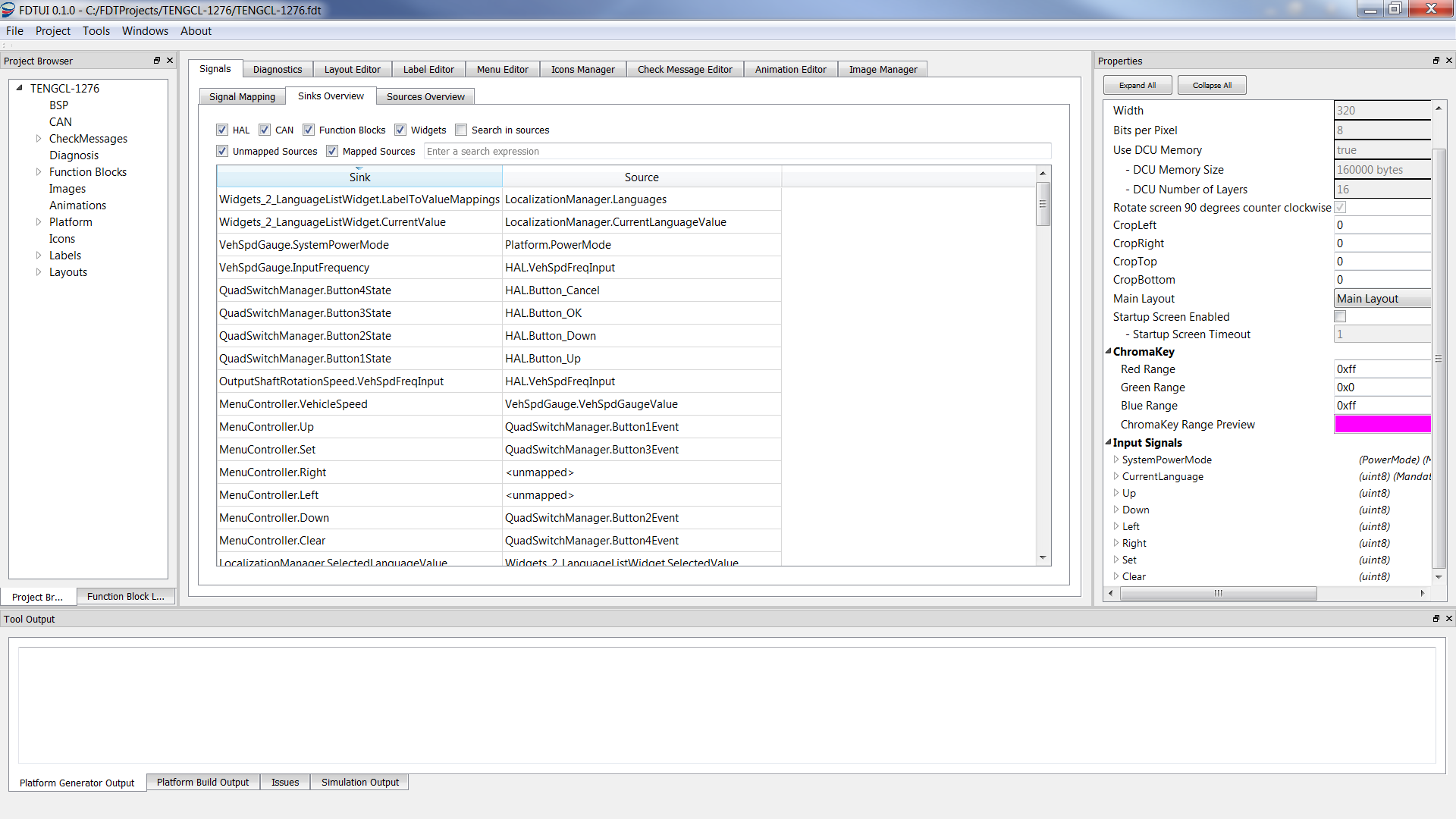
The “**Signal Mapping**” tab shows the connection for the currently selected object in the Project Browser, for example “**Layouts**”:



It shows both the components that the current object is connected to as well as the objects that is connected to it.

##### Sinks Overview

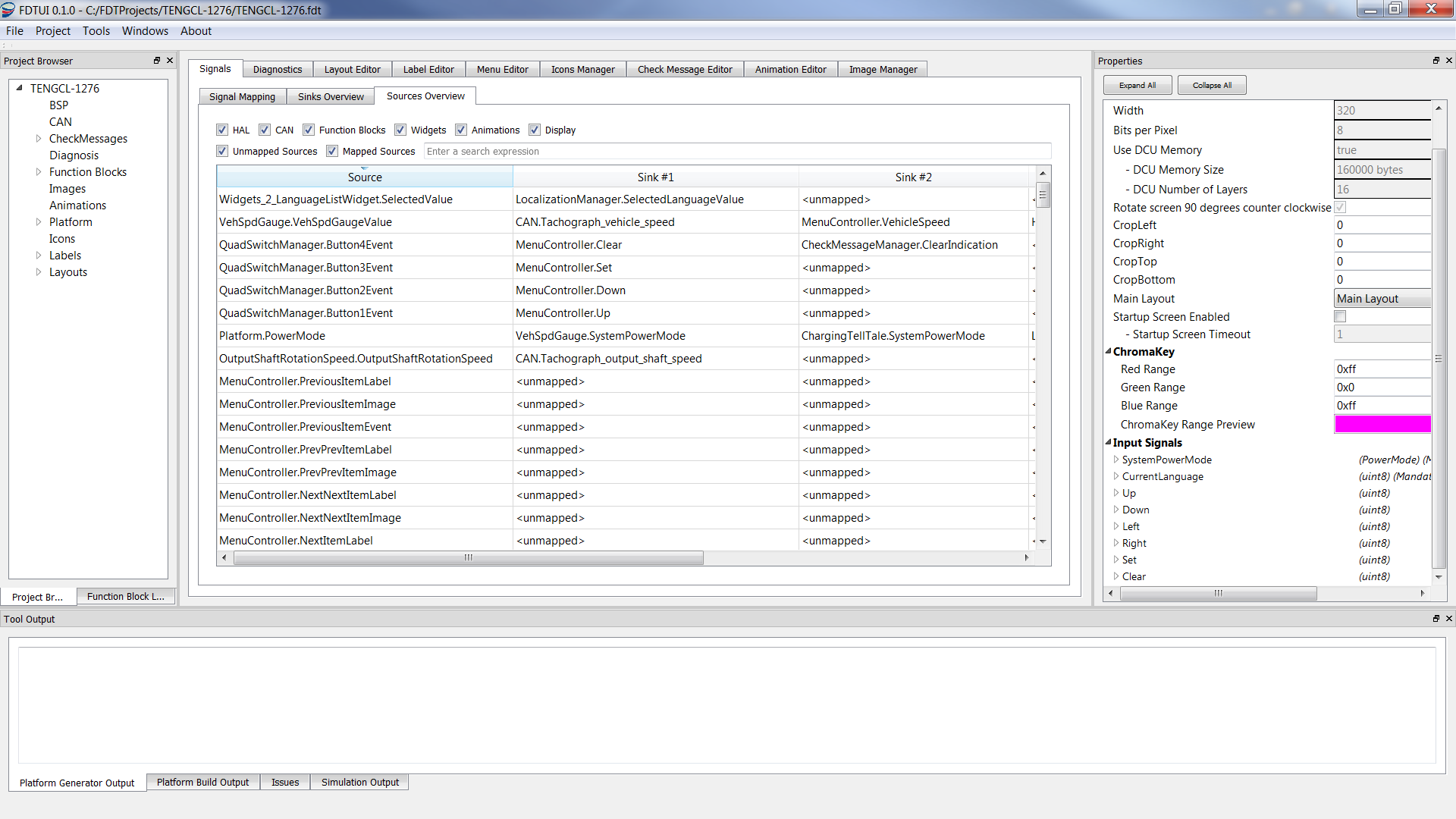
The “**Sinks Overview**” shows all the available sinks and what they are currently connected to.



It is possible to filter the sinks and also to search for specific sinks. It is possible to connect or disconnect sinks either here or in the “**Properties**” window for the object.

##### Sources Overview

The “**Sources Overview**” shows all the available sources and the sinks that are connected to them.



It is possible to filter the sources and also to search for specific sources.

#### Connect Signals for Function Blocks

***Note****: Signals are always connected on the sink side, i.e. where the signal is received. Signal connections in this document are visualized in the following manner:*

***InputSignal******->*** *Module.OutputSignal*

Add the **“LocalizationManager”** function block from the Function Block Library and configure it as follows:

* *LocalizationManager*
  + **SelectedLanguageValue ->** Widgets\_1\_LanguageListWidget.SelectedValue
  + Set Calibration INITIAL\_LANGUAGE\_VALUE = 0x00
  + Add labels to the “languages” label mapping:
    - Expand “Label Mappings”
    - Expand “Language mappings”
    - Click the “Edit Label Mappings” button, and add the labels that shall be used, for example “English” and “Swedish”. Please note that the order is important as this is the order used by the SelectableListWidget when displayed on screen.

Add the **“QuadSwitchManager”** function block from the Function Block Library and configure it as follows:

* *QuadSwitchManager*
  + **Button1State** **->** HAL\_Button\_Up
  + **Button2State** **->** HAL\_Button\_Down
  + **Button3State** **->** HAL\_Button\_OK
  + **Button4State** **->** HAL\_Button\_Cancel

Add the **“VehSpdGauge”** function block from the “Function Block Library” window and configure it as follows:

* *VehSpdGauge*
  + **InputFrequnce** **->** HAL.VehSpdFreqInput
  + **SystemPowerMode** **->** Platform.PowerMode
  + **Needle rest position**: 0x01
  + **Needle max position**: 0x96
  + **VehSpd bias**: 0x00
  + **Max Input Frequency**: 0x13700
  + **Speed Value On Max Input Frequency**: 0x8c
  + **Filter Constant**: 0xc8

Add the **“OutputShaftRotationSpeed”** function block from the Function Block Library and configure is as follows:

* *OutputShaftRotationSpeed*
  + **VehSpdFreqInput** **->** HAL.VehSpdFreqInput
  + **Pulses per rotation:** 0x00
  + **Numerator for ration:** 0x00
  + **Denominator for ratio:** 0x00

Go to “CAN Specification” in the Project Browser and map two mandatory input signals as follows:

* **Tachograph\_output\_shaft\_speed** **->** OutputShaftRotationSpeed.OutputShaftRotationSpeed
* **Tachograph\_vehicle\_speed** **->** VehSpdGauge.VehSpdGaugeValue

There are two more mandatory CAN signals that for now can connect to whatever available as they won’t be used for this tutorial but are needed to be able to generate project:

* **Tachograph\_output\_shaft\_speed\_EP ->**
* **Tachograph\_vehicle\_speed** **->**

Navigate to the properties for the menu. They are found in the “**Project Browser**” under **Layouts-> “the name of the layout containing the menu area”->”the area that contains the menu”**. When selected the menu properties will be shown in the “**Properties**” windows. Map the input signals in the following manner:

* **Up** **->** QuadSwitchManager.Button1Event
* **Down** **->** QuadSwitchManager.Button2Event
* **Set** **->** QuadSwitchManager.Button3Event
* **Cancel** **->** QuadSwitchManager.Button4Event
* **VehicleSpeed ->** VehSpdGauge.VehSpdGaugeValue

Navigate to the “**Menu Editor**” and connect the widgets in the menu in the following manner:

* LanguageListWidget
  + **CurrentValue ->** LocalizationManager.CurrentLanguageValue
  + **LabelToValueMappings ->** LocalizationManager.Languages

***NOTE****: If a specific signal can’t be found, make sure the widget/function block/service is added to the project.*

## Adding an Icon

In the “Icons Manager” tab all icons are added that should be used by the Icon and Dynamic Icon Widgets.

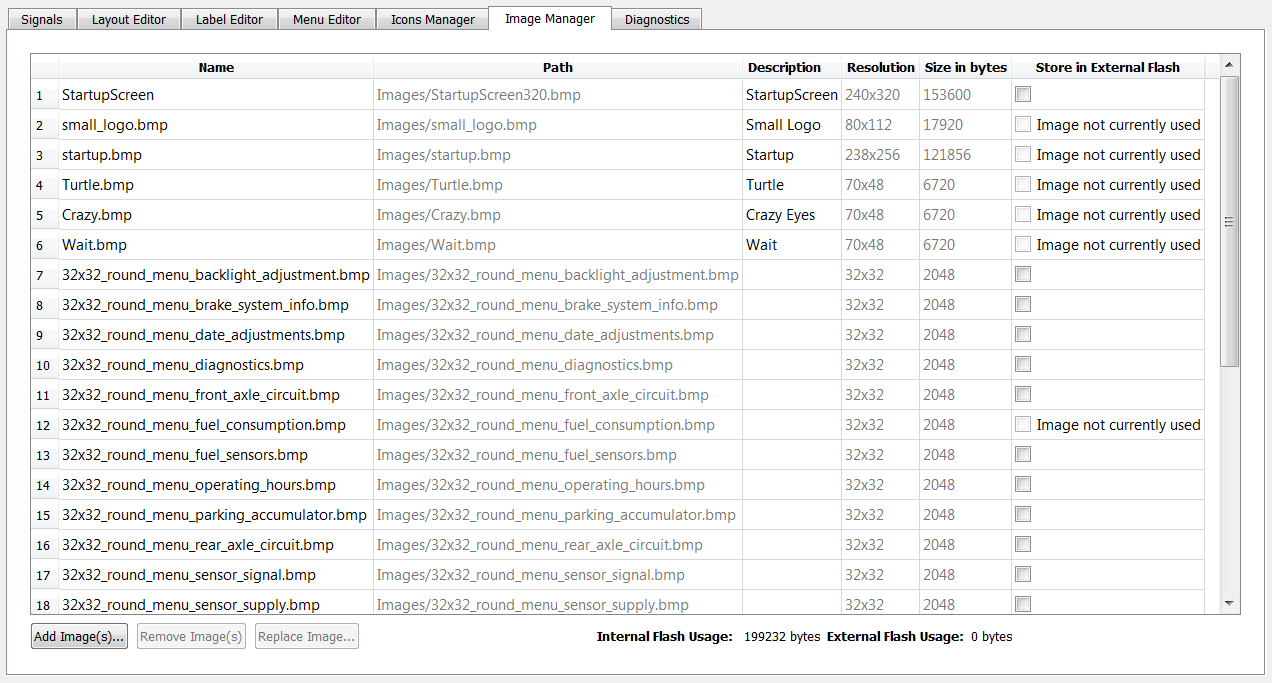
To add an Icon, click the “**Add Icon”** button and select the image representing the icon. Essentially all major image formats are supported, including .bmp, .jpg and .png.

It’s possible to add a description to the icon that will be used instead of the icon name when showing it in the icon chooser for the icons widgets.

To remove an Icon, simply click the red X in front of the row that should be deleted.

## Adding an Image

In the “**Image Manager**” tab all images are added that should be used by Image widgets, Dynamic Image widgets, as well as images that should be used to present menu items.



To add an Image click the “**Add Image(s)**” button and select the image or images that you want to add.

When an image has been added to the Image Manager it’s possible to see the dimensions of it as well as how much memory it uses. If the target hardware has external flash it is also possible to define if an image should be stored on the external flash or not.

If an image is not currently used in the project it will be shown in the “**Store in External Flash**” column and the checkbox in that column will be disabled and the size of that image will not be included when calculating memory usage.

There is also two counters at the bottom of the window that shows memory usages of the images in the Internal and External Memory.

## Adding tell-tales

To add a tell-tale start by adding the function block used to control if the tell-tale should be lit or not. The Charging Switch will be used as an example.

Start by adding the function block, “**ChargingTellTale**”, from the Function Block Library window. Configure it as follows:

* **ChargingIndicatorRequest** -> HAL.Charging\_Indication
* **SystemPowerMode** -> Platform.PowerMode

To connect the output signal click on BSP specification in the Project Browser. This will open the BSP Properties window to the right in the main screen.

Locate “**Charging\_Indication**” and set it to “**ChargingTellTale.ChargingTellTaleValue**”. Generate, build, and run the project in the simulator to verify the functionality.

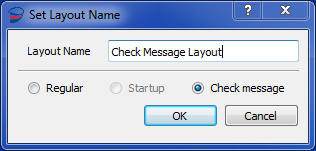
### Connect Gauges

This is similar to adding tell tales. In the chapter regarding the menu, the function block for “**VehicleSpeedGauge**” was added so all that is left is to connect it and this is done in the same way as for tell tales. Locate “**VehSpdGauge**” in the BSP Properties window and connect it to “**VehSpdGauge.VehSpdGaugeValue**”. Generate, build, and run the simulator and verify the functionality.

## Adding Check Messages to the Project

The Check Message support introduces the possibility of showing a modal on top of any regular Layout. A Check Message is mapped to a signal and appears on the user’s screen based on system events that triggers the associated signal.

### Adding a Check Message Layout



In order to add Check Messages to a project, start by adding a new Layout through the **“Layout Editor”** tab. In the Dialog window, enter a Layout name and select the **“Check Message”** radio box. Your Check Message Layout will always appear as the bottommost element in the Layout list. A project can contain either one or no Check Message Layouts.

### Modifying the Check Message Area

A Check Message Layout contains exactly one Area. To configure the appearance of your Check Messages, select the **“Check Message Area”** to activate the associated property widget. When setting the geometry of the Check Message Area, bear in mind that this area will rendered on top of all Layouts in which you want to allow Check Messages to appear.

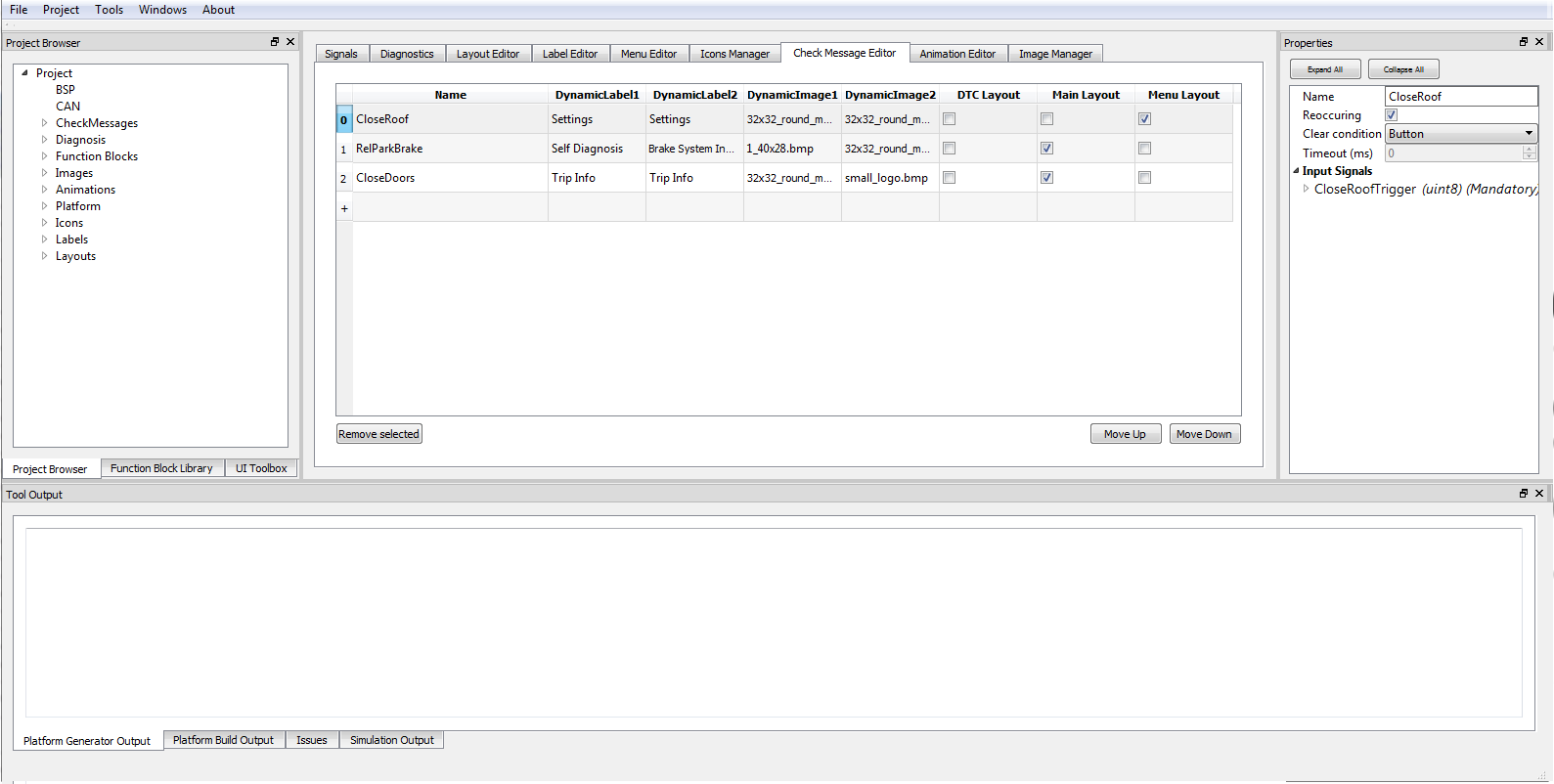
### Adding Widgets to the Check Message Area

The Check Message Area can contain any number of widgets but only the following widgets will be shown in the “**Check Message Editor**”:

* **DynamicLabelWidget**
* **DynamicIconWidget**
* **DynamicImageWidget**

Be aware of the hardware restrictions of the target platform. Allowing a Check Message to appear in a Layout means that the DCU Layer usage of the Check Message Layout is added on top of the DCU Layer usage of the Layout itself. The Layer usage counter in the Layout Preview screen will keep track of these restrictions for you.

### Creating a Check Message



The design of Check Messages is separated from the configuration of the Check Messages themselves. In FDT, navigate to the **“Check Message Editor”** tab to start configuring Check Messages. Start by double-clicking in the **“Name”** column in the bottommost row to create a new Check Message and assigning a name to it. The name is only used for the designer’s reference and never visible on the platform. The message ordering in the table reflects a priority ordering. If multiple Check Messages are eligible to be shown at the same time, the topmost Check Message will be stacked on top of other messages with lower priority.

The columns after the Check Message name represents a mapping between all widget sources in the Check Message Area and the desired resources created by the cluster designer (labels, icons and images). This enables the static Check Message design to display variable content in different contexts.

The last column group creates a check box for every Layout in the project. A checked box allows the Check Message to be shown in the corresponding Layout.

The other properties of a Check Message is available through the Check Message Property Widget. Select a row in the **“Check Message Editor”** to bring up its Property Widget. The properties are as follows:

* **Reoccurring –** Determines whether a Check Message will reappear if already cleared once. This property applies to individual messages, not per Layout. This property is not applicable to the Trigger Clear Condition, which will always appear along with the signal.
* **Clear condition –** Sets the way a Check Message is cleared. It can be cleared by a user button press, a fixed timeout or by the signal trigger itself being reset.
* **Timeout –** Only applicable in conjunction with the “Timeout” Clear Condition.
* **Input signal –** The signal used to trigger the Check Message.

## Save the Project

Choose **File­ ->** **Save** in the menu to save the project.

### Generate and Build the Project

Choose **Project** **-> Generate** in the menu, or use the short command: **Ctrl+G**

This will generate all the needed source and header files needed to build the chosen target and copy them to the project folder.

To build the project click **Project->Build** in the menu, or use the short command: **Ctrl+B**

***NOTE:*** *It’s important to make sure each step is successful before continuing, i.e. that the code generation was completed successfully before trying to build. Use the Tool Output Window (described in the next section) for this.*

#### Tool Output

When generating, building and running the project, FDT will produce a lot of output. This is shown in four tabs on the Tool Output in the lower part of the main window.

* **Platform Generator Output**: Shows the output from the platform generator when generating the platform code.
* **Platform Build Output**: Shows the compiler output when building the platform code generated by the platform generator.
* **Issues**: If any warnings and/or errors occurs while generating and building the project they will show up here. If the errors are so severe that the generating or build fail, this tab will automatically be focused.
* **Simulation Output**: Shows the PCIC output when running the project and the simulator.

The tabs will automatically be focused depending on what is being done at the moment.

***Note:*** *When the build fails when building for target the error message is most likely found in the Platform Output tab.*

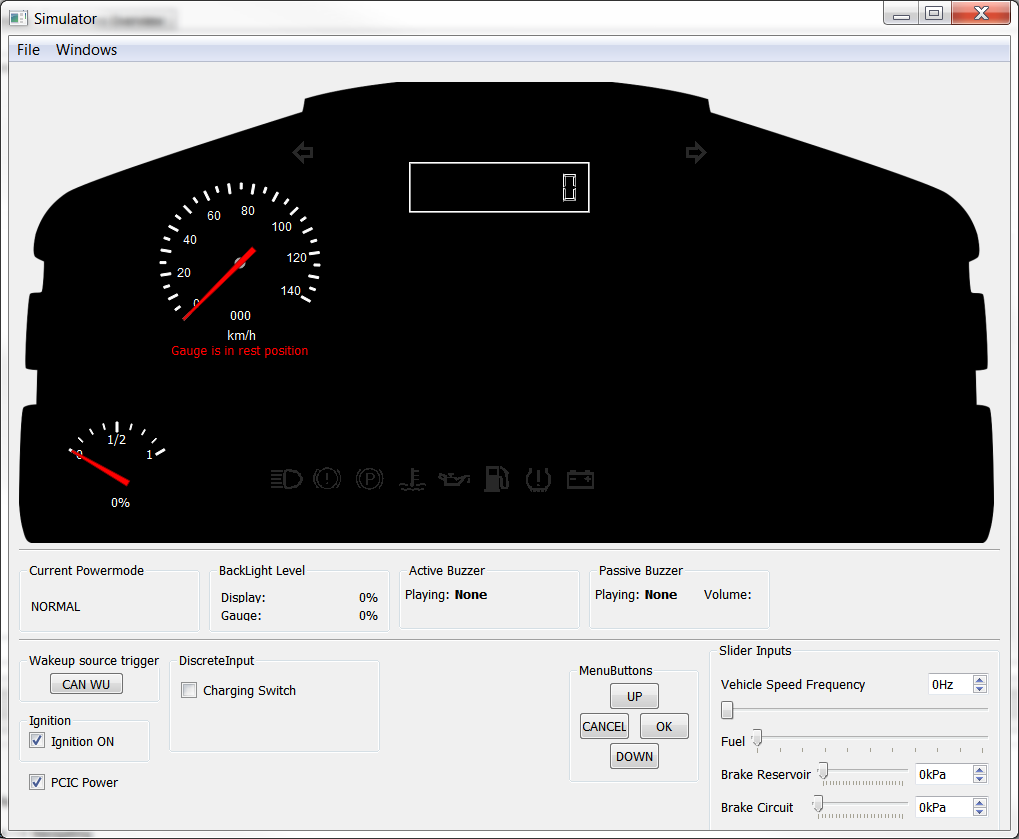
# The Simulator

To be able to test and verify the project easily FDT includes a simulator.

To start it, first generate and build the project for “**Simulator”** target, then choose **Project**->**Run Simulator** (or use the short command **Ctrl+R**)to start the simulator.

Click **“PCIC Power”** and then **“Ignition On”.** This starts the PCIC which is the newly generated and built software. Notice that there are output written to the tool output window in FDT.

If the PCIC is successfully started the menu will appear and the different input controls will be enabled.



The layout display is shown in a separate window:



See next section for a description of each input control.

## Input Controls

### Discrete Inputs

These are controls to simulate on/off signals that usually comes from HAL.

### Menu Buttons

Simulates the buttons used to navigate the menu.

### Slider Inputs

These are slider controls that is used to simulate values within a certain span. They are used for simulating vehicle speed, fuel level, brake pressure, etc.

## CAN Input

The simulator support CAN messages. This means that it is possible to use a CAN message generator, such as BusMaster, to send CAN messages to the simulator to trigger certain behaviors.

## Configure the Simulator

### Add Tell Tales to the Simulator

One of few configurations available for the simulator is the adding of the tell tales used. These are added in two steps:

1. Add the Tell Tale bmp file to the folder “telltales” located in the simulator folder
2. Add the telltale to the “simulator\_cfg.cml”:
   1. <TellTale Index="0x4E" BitmapPath="telltales/M01.bmp" ActiveColor="Red" />

Index is found in the BSP specification and “ActiveColor” is the color that the tell-tale should have when lit.

## Persistent Memory (Simulated EEPROM)

There is a function called persistent memory in the simulator which can be used to verify EEPROM reading and writing when running the simulator. Click **Windows->Persistent Memory** menu to revel the persistent memory window.

***NOTE****: The values in the persistent memory window isn’t stored when the simulator is closed.*

# Function Blocks

Function Blocks are code segments that are used to perform a specific task and generate an output that can be used to set signals or display information to the user.

## BrakePressureLowTellTale

### Purpose

The purpose of the function block is to read the reservoir and circuit pressures and output if a warning should be active or not.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** **->** Platform.PowerMode
* **FrontReservoirPressure** **->** HAL.FrontReservoirPressure
* **RearReservoirPressure ->** HAL.RearReservoirPressure
* **ParkBrakeReservoirPressure** **->** HAL.ParkBrakeReservoirPressure
* **FrontCircuitPressure** **->** HAL.FrontCircuitPressure
* **RearCircuitPressure** **->** HAL.RearCircuitPressure

### Output signals

* **BrakePressureLowIndicatorValue** – Indicates if the warning signal is active or not.

### Calibration Parameters

None

### Configuration Parameters

None

## ChargingTellTale

### Purpose

The purpose of the function block is to read a signal and output if the charging indicator is active or not.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPower** **->** Platform.PowerMode
* **ChargingIndicatorRequest** **->** HAL.Charging\_Indication

### Output signals

* **ChargingTellTaleValue** – Indicates if the warning signal is active or not.

### Calibration Parameters

None

### Configurations

None

## EngineOilPressureTellTale

The purpose of this function block is to read a signal and output if the warning indication should be active or not.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** **->** Platform.PowerMode
* **OilPressureWarningRequest** **->** CAN.oil\_pressure\_warning

### Output signals

* **OilPressureWarningIndication** – Indicates if the warning signal is active or not.

### Calibration Parameters

None

### Configurations

None

## FuelConsuption

### Purpose

The purpose of the function block is to read input signals from CAN and output filtered and edited values.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **InstantaneousConsumption** -> CAN.instantaneous\_fuel\_economy
* **FuelRate** -> CAN.fuel\_rate
* **ResetSignal** -> SignalTriggerWidget.SignalTriggerSource

### Output signals

* **TotalFuelConsumption** – The total fuel consumption since the vehicle left the factory.
* **TripFuelConsumption** – The fuel consumption since the trip was last reset.
* **FilteredInstantaneousFuelConsumption** – The current fuel consumption filtered to show smother changes in the fuel consumption.

### Calibration Parameters

* **Total Fuel Consumption** – The total fuel consumption in liters.
* **Total Fuel Consumption Ticks** – The fuel consumption since the last liter was written.

### Configurations

* **Filter Period** – A value between 0 and 10000, higher value means the historic average is more important, lower value means latest value is more important.

## FuelLevel

### Purpose

The purpose of the function block is to read a fuel level signal and output a filtered value as well as a level low indicator when needed

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **FuelLevel** -> HAL.FuelLevel

### Output signals

* **FuelLevelGaugeValue** -> The filtered fuel level value.
* **FuelLevelWarningIndication** -> Indicates if the fuel level warning should be active or not.

### Calibration Parameters

* **Warning threshold in percent** – When the fuel level goes below this percentage, the indicator is lit.
* **Fuel warning hysteresis parameter** – When the fuel level reaches above the threshold by this amount, the fuel warning indicator turns off. This is to prevent the indicator to flicker if the level is around the warning threshold.

### Configurations

* **Filter Coefficient** -- A value between 0 and 10000, higher value means the historic average is more important, lower value means latest value is more important

## HighBeamTellTale

### Purpose

The purpose of the function block is to read a signal and set a value to indicate if the high beam tell-tale should be active or not.

### Prerequisites

None

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **HighBeamIndicatorRequest** -> CAN.high\_beam

### Output signals

* **HighBeamTellTaleValue** – Indicates if the high beam tell-tale should be active or not.

### Calibration Parameters

None

### Configurations

None

## IlluminationManager

### Purpose

The purpose of the function block is to hold and set the illumination level of the gauges and display based on the current Illumination mode (daytime or nighttime).

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **IlluminationMode** -> CAN.back\_lamp
* **IlluminationLevel** -> CollumnWidget.SelectedValue

### Output signals

* **GaugeIlluminationLevel** – The illumination level of the gauges
* **DisplayIlluminationLevel** – The illumination level of the display
* **CurrentIlluminationLevel** – Used by the CollumnLevelWidget as input signal to know the current level.

### Calibration Parameters

* **Display illumination level**  – PWM illumination output for display illumination level 1 - 5
* **Gauge illumination level** – PWM illumination output for gauge illumination level 1 - 5
* **Current illumination level** – The current illumination level

### Configurations

None

## LocalizationManager

### Purpose

The purpose of the function block is to hold and send out the current selected language as well as hold a container of labels representing the available languages.

### Prerequisites

None

### Input signals

* **SelectedLanguageValue** -> SelectableListWidget.SelectedValue

### Output signals

* **CurrentLanguageValue** – The current language
* **Languages** – The available languages

### Calibration Parameters

* **Selected Language** - Initial language value used in the menu at startup

### Configurations

* **Language mappings** – A list of labels representing the available languages

## OdoAndTripDisplayHandler

### Purpose

The purpose of the function block is to let the user decide if information regarding the trip distance and ODO should be shown or if information about trip fuel consumption and total fuel consumption should be shown.

### Prerequisites

None

### Input signals

* **ShowNext\_Trigger1** -> The signal that should trigger a change in displayed information
* **ShowNext\_Trigger2** -> The signal that should trigger a change in displayed information
* **OdoValue** ->ODOTrip.CurrentODOValue
* **TripValue** -> ODOTrip.CurrentTripValueA
* **InstantaneousFuelConsumption** -> FuelConsumption.FilteredInstantaneousFuelConsumption
* **TripFuelConsumption** -> Fuelconsumption.tripFuelconsumption

### Output signals

* **TripFieldText** – The text string containing the trip fuel or distance information
* **OdoAndInstFuelFieldText** – The text string containing the ODO or fuel consumption info

### Calibration Parameters

None

### Configurations

None

## OdoRuntimeHandler

### Purpose

The purpose of the function block is to output ODO or total runtime on e.g. a segment display depending on an input signal.

### Prerequisites

Ignition must be on.

### Input signals

* **ShowNext** -> The signal that should trigger a change in the displayed info
* **RuntimeValue** ->
* **OdoValue** -> ODOTrip.CurrentODOValue

### Output signals

* **Value** – The current value to display
* **ShowDot** – Defines if the decimal dot should be shown or not. For Segment displays
* **ShowHourGlass** – Defines if the hourglass should be shown or not. For Segment displays

### Calibration Parameters

None

### Configurations

None

## ODOTrip

### Purpose

The purpose of the function block is to keep track of the ODO, the total distance that the vehicle has been driven.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **VehSpdFrequency** -> HAL.VehSpdFreqInput
* **TripResetA** -> SignalTriggerWidget.SignalTriggerSource
* **TripResetB** -> SignalTriggerWidget.SignalTriggerSource

### Output signals

* **CurrentODOValue** – The current ODO value
* **CurrentTripAValue** – The current Trip A distance
* **CurrentTripBValue** – The current Trip B distance

### Calibration Parameters

* **NumberOfRestPulses** – Number of pulses not counted in full km
* **NumberOfRestPulsesTripA** – Number of pulses not counted in full hundreds meters
* **NumberOfRestPulsesTripB** – Number of pulses not counted in full hundreds meters
* **Number** of pulses per kilometer -- Number of pulses per kilometer
* **Current ODOMeter value** – Storage areas for ODO. There are 5 areas that are used and the reason for this is that the EEPROM only can handle a limited number of writings to a certain memory area before it brakes and by using 5 different areas the wear of the EEPROM area is reduced. When ODO is read from the EEPROM all five areas are read and the highest value is the latest stored ODO.
* **Current TripA value** – The current Trip A distance
* **Current TripB value** – The current Trip B distance

### Configurations

None

## OuptutShaftRotationSpeed

### Purpose

The purpose of the function block is to send out the current speed on CAN

### Prerequisites

None

### Input signals

* **VehSpdFreqInput** -> HAL.VehInputSpdFreqInput

### Output signals

* **OutputShaftRotationSpeed** – The speed frequency value

### Calibration Parameters

* **Pulses per rotation** – Number of pulses from sensor per rotation.
* **Numerator for ratio** – factor used to translate the value from HAL to CAN.
* **Denumerator for ratio** – factor used to translate the value from HAL to CAN.

### Configurations

None

## ParkBrakeIndicator

### Purpose

The purpose of the function block is to set a signal to indicate if the park brake indicator is active or not and also set a warning signal and play a warning sound if the park brake is active and the vehicle is rolling.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **VegSpd** -> VehSpdGauge.VehSpdGaugeValue
* **ParkBrakeSwitch** -> CAN.parking\_brake\_lamp

### Output signals

* **ParkBrakeIndicatorCheckMessageActive** – Indicates if the check message warning message should be displayed.
* **ParkBrakeIndicatorValue** – Indicates if the park brake indicator should be active or not.

### Calibration Parameters

None

### Configurations

None

## QuadSwitchManager

### Purpose

The purpose of the function block is to trigger events when mapped buttons are pressed or released

### Prerequisites

None

### Input signals

* **Button1State** -> HAL.Button\_Up
* **Button2State** -> HAL.Button\_Down
* **Button3State** -> HAL.Button\_OK
* **Button4State** -> HAL.Button\_Cancel

### Output signals

* **Button1Event** – Triggered when the state of Button1 is changed
* **Button2Event** – Triggered when the state of Button2 is changed
* **Button3Event** – Triggered when the state of Button3 is changed
* **Button4Event** – Triggered when the state of Button4 is changed

### Calibration Parameters

None

### Configurations

None

## TPMS

### Purpose

The purpose of the function block is to present the tire pressure for all tires as well as setting a warning signal if the warning signal is active in the read CAN message. The information is received over CAN and each message includes one tire at the time so it takes 10 runs of the function block to complete all tires.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **TPMSTireLocation** -> CAN.tire\_location
* **TPMSTirePressure** -> CAN.tire\_pressure
* **TPMSTirePressureWarning** -> CAN.tire\_pressure\_warning

### Output signals

* **TPMSFrontLeftTireValue** – The value of the front left tire
* **TPMSFrontRightTireValue** – The value of the front right tire
* **TPMSMiddleAxleLeftOuterTireValue** – The value of the middle axle left outer tire
* **TPMSMiddleAxleLeftInnerTireValue** – The value of the middle axle left inner tire
* **TPMSMiddleAxleRightOuterTireValue** – The value of the middle axle right outer tire
* **TPMSMiddleAxleRightInnerTireValue** – The value of the middle axle right inner tire
* **TPMSRearAxleLeftOuterTireValue** – The value of the rear axle left outer tire
* **TPMSRearAxleLeftInnerTireValue** – The value of the rear axle left inner tire
* **TPMSRearAxleRightOuterTireValue** – The value of the rear axle right outer tire
* **TPMSRearAxleRightInnerTireValue** – The value of the rear axle right inner tire
* **TPMSIndicatorValue** – Indicates if the tire pressure low warning should be active.

### Calibration Parameters

* **TPMS Installed** – Determines if TPMS info shall be displayed in menu. 0x00 = disabled, 0x01 = enabled

### Configurations

None

## TurnIndication

### Purpose

The purpose of the function block is to activate the run indicator tell tales and sound.

### Prerequisites

None

### Input signals

* **RightTurnIndication** -> CAN.vehicle\_right\_turn\_lamp
* **LeftturnIndication** -> CAN.vehicle\_left\_turn\_lamp

### Output signals

* **TurnRightTellTaleValue** – Indicates if the turn right tell-tale and sound should be activated
* **TurnLeftTellTaleValue** – Indicates if the turn left tell-tale and sound should be activated

### Calibration Parameters

* **Sound delay in 10ms increments** – The telltale will be delayed by this many runnable periods after indication

### Configurations

None

## UreaLevelMeter

### Purpose

The purpose of the function block is to output the uera level.

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **CatalystTankLevel** -> CAN\_catalyst\_tank\_level

### Output signals

* **Segment0** – Indicates if the first segment in the level meter should be lit.
* **Segment1** – Indicates if the second segment in the level meter should be lit.
* **Segment2** – Indicates if the third segment in the level meter should be lit.
* **Segment3** – Indicates if the fourth segment in the level meter should be lit.

### Calibration Parameters

None

### Configurations

None

## VehSpdGauge

### Purpose

The purpose of the function block is to present the vehicle speed in km/h

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **InputFrequency** -> HAL.VehSpdFreqInput

### Output signals

* **VehSpdGaugeValue** – The vehicle speed in km/h

### Calibration Parameters

* **Max Input Frequency (24.8 FPF)** – The maximum value for the input frequency
* **Speed Value On Max Input frequency** – The value for vehicle speed given maximum input frequency
* **Filter Constant** – Filter constant for the first degree filter

### Configurations

* **Needle Rest Position** – Value to set when requesting rest position
* **Needle max position** – Max value for the needle position
* **VehSpd bias** – Vehicle speed bias

## WarningIconFields

### Purpose

The purpose of the function block is to output icons depending on incoming signals. This function block is used by the Dynamic Icon Widget.

### Prerequisites

None

### Input signals

* **ChargingIndication** -> HAL.Charging\_Indication
* **ParkbrakeIndication** -> ParkBrakeIndicator.ParkBrakeIndicatorValue
* **BrakePressureLowIndicator** -> BrakePressureLowIndicator.BrakePressureLowIndicatorValue
* **EngineOilPressureIndication** -> EngineOilPressureTellTale.OilPressureWarningIndication

### Output signals

* **Icon1ID** – The ID of the icon to be displayed
* **Icon2ID** – The ID of the icon to be displayed
* **Icon3ID** – The ID of the icon to be displayed
* **Icon4ID** – The ID of the icon to be displayed

### Calibration Parameters

None

### Configurations

Icons – Mapping of the signals to the responding icon

## WaterTempHighIndicator

### Purpose

The purpose of the function block is to activate a signal if the water temperature gets to high

### Prerequisites

Ignition must be on.

### Input signals

* **SystemPowerMode** -> Platform.PowerMode
* **WaterTempHighIndication** -> CAN.water\_warning

### Output signals

* **WaterTempHighTelltaleValue** – Indicates if the water temp high warning is active or not

### Calibration Parameters

None

### Configurations

None

# Creating a Function Block

The FDT tool provides a guide to create own function blocks in the main menu under **Tools** -> **Create Function Block…**

The “Create New Function Block” wizard consists of a number of screens in which the function block is configured. When finished, the wizard will generate a Function Block skeleton with examples on how to use the different configured parameters.

## Basic Parameters

The first parameter to enter is the name of the function block, it must be a single word, without any special characters. The new function block will be added in the root folder of the Function Library in a subfolder named after the function block.

The initial delay and periodicity parameters apply to the runnable created by the guide in the function block template. They determine how long after startup the runnable will be executed for the first time and at the periodicity of subsequent executions.

## Configure Input Signals

The input signals define which external data is required for the function block to perform its duties. Each signal is defined by four parameters; **Name**, **DataType**, **Notify** and **Mandatory**.

* **Name** – should contain no spaces or special characters and begin with an alphabetical character.
* **DataType** – should be a platform defined basic (e.g. uint32, sint16…) or complex (e.g. dateTime\_t, FPF\_u32\_8\_Hz…) datatype.
* **Notify** – indicates whether the function block would like to receive a notification from GCL when the signal value is updated. (Only supported on CAN signals and signals from other Function Blocks)
* **Mandatory** – indicates that an input signal is mandatory, i.e. it has to be mapped towards a signal source otherwise the project generation will fail.

## Configure Output Signals

The output signals define which signals are emitted by the function block. Output signals only have two parameters; **Name** and **DataType**. These parameters are the same as for the Input Signal definitions.

## Setup Configuration Parameters (design-time)

Configuration Parameters are presented in the FDT tool with the function block and allows for design-time configuration of the function block’s behavior. They are defined by four parameters; **Name**, **DataType**, **Descriptive** **Name** and **Description**.

* **Name** – is translated to a variable name that can be used in the function block, so this parameter should follow standard C variable naming rules.
* **Data Type** – should be a basic datatype (e.g. uint32, sint16 …)
* **Array Length** – a number between 1 to 255 defining the array size
* **Descriptive Name** – is used in the tool to give users an understanding of what the parameter is for, this should be a fairly short description.
* **Description (optional)** – a longer description to provide the details about the parameter, e.g. what different values mean, or a range.

## Setup Calibration Parameters (run-time)

The calibration parameters are almost identical to the Configuration Parameters with the only difference being that they are given a default value in the FDT tool which is set on startup of the IPC and can later on be manipulated by diagnostics tools during runtime. The four parameters are the same as for the configuration parameters with the only difference that the **Name** should be in ALL CAPS and contain only alphabetical characters and underscores. A good practice is to prefix them with the function block name to make sure that they are unique in the platform, e.g. VEHSPDGAUGE\_MAXFREQUENCY.

## LabelMappings

Label mappings are used when a function block shall control a text field on the display using labels defined in FDT. Each of these mappings will enable the user to map any number of labels to the function block which can then be used to send the label’s unique ID to a text field widget on the display.

Each mapping defined on this screen can contain any number of mappings, so they should be defined according to usage, for example: “LanguageLabels” and “UnitOfMeasurementLabels”.

* **Name** – used in the function block to reference the specific mapping
* **Descriptive Name** – is used in the tool to give users an understanding of what the parameter is for, this should be a fairly short description
* **Description (optional)**– a longer description of what the mapping is used for

## Icon Resources

Enables the function block to use Icon resources defined in FDT. Each resource name defined can then be used in the function block to get the Icon ID mapped by the user.

## Review Function Definition

The last step before the Function Block is created is a review pane where all parameters can be inspected. When approved, click “Finish” to generate the Function Block.

## Adding Tasks/Signals/Parameters after running the wizard

The wizard gives an understanding of how the different parameters are used by the IC platform and how they can be used in the Function Block code. However, it is not always known beforehand exactly which tasks, signals or parameters that the function block need so this chapter describes how to make adjustments after the wizard is completed.

### Add a Runnable

**Step 1:** Add a new <Task> element in the <FunctionDefinition><Tasks> section.

<FunctionDefinition Name=”MyFunctionBlock”>

<Tasks>

<Task>

<InitFunction>MyFunctionBlock\_Init</InitFunction>

<Runnable>MyFunctionBlock\_Run</Runnable>

<StartupDelay>10</StartupDelay>

<Periodicity>20</Periodicity>

</Task>

**<Task>**

**<InitFunction>MyFunctionBlock\_AnotherInit</InitFunction>**

**<Runnable>MyFunctionBlock\_RunSomethingElse</Runnable>**

**<StartupDelay>15</StartupDelay>**

**<Periodicity>20</Periodicity>**

**</Task>**

</Tasks>

….

</FunctionDefinition>

**Step 2:** The function names in the InitFunction and Runnable elements must be defined in a source file somewhere in the **src** subdirectory. (This is true also if the function names in the template Task are modified.) They should have the function prototype:

**void FunctionName(void);**

**Step 3:** Define the initialization and runnable code in the named functions.

### Add an Input Signal

**Step 1:** Add a new <Signal> element in the <FunctionDefinition><SignalInterface><InputSignals> section.

<FunctionDefinition Name=”MyFunctionBlock”>

…

<SignalInterface>

<InputSignals>

**<Signal Name=”MyInputSignal” DataType=”uint16” UpdateNotification=”True” Mandatory=”True”/>**

</InputSignals>

</SignalInterface>

</FunctionDefinition>

**Step 2:** If not already included, include the file GCL\_<functionblockname>.h in the source file. The header file is provided by the tool and does not exist until after the IC software is generated.

**Step 3:** Read the signal from GCL by calling the function GCL\_Read\_<FunctionBlockName>\_<SignalName>(<DataType>\* value, uint8\* status);

/\* Prototype \*/

void GCL\_Read\_MyFunctionBlock\_MyInputSignal(uint16\* value, uint8\* status);

/\* Usage \*/

uint16 value;

uint8 status;

GCL\_Read\_MyFunctionBlock\_MyInputSignal(&value, &status);

**Step 4:** If the UpdateNotification attribute is set to “True”, add an indication function to the function block source code. The indication function should be named <FunctionBlockName>\_<SignalName>\_Indication()

/\* Prototype \*/

void MyFunctionBlock\_MyInputSignal\_Indication(void);

/\* Usage \*/

void MyFunctionBlock\_MyInputSignal\_Indication(void)

{

/\* Add code that should be run when the signal value changes in the IC or set a flag that makes the runnable re-read the signal when next executed \*/

}

### Add an Output Signal

To add another output signal to the function block:

**Step 1:** Add a new <Signal> element in the <FunctionDefinition><SignalInterface><OutputSignals> section.

<FunctionDefinition Name=”MyFunctionBlock”>

…

<SignalInterface>

<OutputSignals>

**<Signal Name=”MyOutputSignal” DataType=”uint32” UpdateNotification=”False” />**

</OutputSignals>

</SignalInterface>

…

</FunctionDefinition>

**Step 2:** If not already included, include the file GCL\_<functionblockname>.h in the source file. The header file is provided by the tool and does not exist until after the IC software is generated.

**Step 3:** Write the signal to GCL by calling the function GCL\_Write\_<FunctionBlockName>\_<SignalName>(<DataType> value);

/\* Prototype \*/

void GCL\_Write\_MyFunctionBlock\_MyOutputSignal(uint32 value);

/\* Usage \*/

uint32 value = 0xAABBCCDD;

GCL\_Write\_MyFunctionBlock\_MyOutputSignal(value);

### Add a Configuration Parameter

**Step 1:** Add a new <ConfigurationParameter> element to the <FunctionDefinition><ConfigurationParameters> section.

<FunctionDefinition Name=”MyFunctionBlock”>

…

<ConfigurationParameters>

**<ConfigurationParameter DescriptiveName=”My configuration parameter” ParameterName=”u8MyConfigurationParameter” Description=”My very own configuration parameter which sets important stuff” DataType=”uint8” />**

</ConfigurationParameters>

…

</FunctionDefinition>

**Step 2:** Declarean **extern const** variable with the ParameterName in the source code. The definition with the value configured in the tool will be added in an automatically generated configuration source file which is linked in at compile-time.

/\* In the source file \*/

extern const uint8 u8MyConfigurationParameter;

/\* In the generated configuration file \*/

const uint8 u8MyConfigurationParameter = 150;

**Step 3:** Use the configuration parameter in the source code.

void MyFunctionBlock\_Run()

{

…

uint16 value = u8MyConfigurationParameter << 5;

…

}

### Add a Calibration Parameter

**Step 1:** Add a new <CalibrationParameter> element to the <FunctionDefinition>< CalibrationParameters> section.

<FunctionDefinition Name=”MyFunctionBlock”>

…

<CalibrationParameters>

**<CalibrationParameter DescriptiveName=”My calibration parameter” ParameterName=”MYFUNCTIONBLOCK\_CALIBRATIONPARAMETER” Description=”My very own calibration parameter which sets important stuff” DataType=”uint16” />**

</CalibrationParameters>

…

</FunctionDefinition>

**Step 2:** If not already included, include the “CAL.h" header file in the source code.

**Step 3:** Use the calibration parameter in the source code by calling the function CAL\_Read<8|16|32> where 8, 16 or 32 depends on the data type of the calibration parameter.

/\* Prototype \*/

CAL\_STATUS CAL\_Read16(uint16 u16ParameterId, uint16\* value);

/\* Usage \*/

void MyFunctionBlock\_Run()

{

uint16 calibrationParameter;

CAL\_Read16(MYFUNCTIONBLOCK\_CALIBRATIONPARAMETER, &calibrationParameter);

…

}

# Widgets

To populate the menu and add functionality to the display there are a number of widgets available. They are used for displaying different data and gives the developer a variety of possibilities to configure the layout.

***Note:*** *When a color display is used, and if the display uses “Hardware Rendering” it’s also possible to configure the color of the widgets that has support for color. This is done by selecting a widget and specifying the RGB values of a desired color in the widget property view.*

*It is also possible to use animations when hardware rendering is used. Animations can be tied to a widget by choosing a defined animation in one of the combo boxes that are shown in the widgets property view. Please note that only widgets that are hardware rendered (ImageWidget, DynamicImageWidget) can use animations.*

*Furthermore it’s possible to define a chroma key color (a color used by the DCU to simulate transparency) when using “Hardware Rendering”. This can be used on images and as for example backgrounds of widgets so that a layer beneath it is shown even if the widgets are overlapping. The chroma key color is specified in the “Layout” in the project browser.* ***Please note:*** *The color specified as chroma key color will never be visible.*

## Label widget

This is the simplest widget that consists of a static text.

It does not allocate any layer on hardware rendered platforms.

It has the following configurable options:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Label** – The label, created in Label Editor, which should be shown by the widget.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Icon widget

This widget is used to display an icon or picture in the menu area.

It does not allocate any layer on hardware rendered platforms.

It has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Icon** – The icon, defined in Label Editor, which should be shown by the widget.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Image Widget

This widget is used to display an image.

It allocates one layer on hardware rendered platforms.

If an animation is selected the animation properties will be visible. Depending on the settings for the animation in the Animation Editor, some of the properties here might be disabled.

It has the following attributes:

* **Name**
* **X** – The x position based on the area it’s placed in.
* **Y** – The y position based on the area it’s placed in.
* **Width** – *Not Editable*. Set to the width of the image.
* **Height** – *Not Editable*. Set to the height of the image.
* **Boxed** – Defines if the widget should have a border or not.
* **Image** – The image to be displayed by the widget.
* **Animation1** – First animation slot.
  + **Use Triggers 1** – Select if triggers should be used to start the animation.
    - **Behaviour on Load** – Specifies the behavior of the animation when the layout is loaded.
      * **Do Nothing** – Does nothing
      * **Run One Cycle** – Runs on cycle
      * **Show Default Frame** – Shows the default frame set for the animation in the Animation Editor.
    - **Behaviour on Trigger Active** – Specifies the behavior of the animation when the trigger goes to active.
      * **Start from Beginning** – Specifies if the animation should start from the beginning or if it should start from where it’s at.
    - **Behaviour on Trigger Inactive** – Specifies the behavior of the animation when the trigger goes to inactive.
      * **Stop** – Stops immediately when the trigger is inactive
      * **Stop at end of cycle** – Stops when the animation reaches the end of the cycle.
      * **Stop at Default Frame** – Stops at the default frame specified for the animation in the Animation Editor.
      * **Continue** – The animation continue to run until the Layout changes.
* **Animation2** – Second animation slot.
  + **Use Triggers 2** – Select if triggers should be used to start the animation.
    - **See the settings under Use Triggers 1**

## Dynamic Icon widget

This container can be used to show different icons in the same area depending on the status of different input signals.

Currently the dynamic icon widget works together with the function block “WarningIconFields”. One dynamic icon widget can only show one icon at a time so the function block in question is actually connected to four dynamic icon widgets to achieve a desired design.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **IconID to display** – The icon source that should be shown by the widget.

## Blockbar widget

This widget can be used to display a value in the form of a block bar.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* Name
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Spacing** – The spacing between the blocks
* **Blocks** – The number of blocks in the block bar
* **Min** – The lower limit of the block bar.
* **Max** – The upper limit of the block bar.
* **Datatype** – The data type of the data used as input for the block bar. This value is used to filter available sources that Signal Sink can be connected to.
* **Signal Sink** – The source of the data shown by the block bar.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Column Level widget

This widget can be used as an input for the user to set for example the level of illumination on the screen.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width of the widget
* **Height** – The maximum height of the widget.
* **Spacing** – the number of pixels between the blocks
* **Blocks** – Number of blocks to represent the level
* **Min** – The input value when the widget should show its minimum value
* **Max** – The input value when the widget should show its maximum value
* **Min block height** – The height of the lowest block
* **Max block height** – The height of the highest block
* **CurrentValue** – The input value for the widget
* **SelectedValue** – The source value for other widgets or function blocks to use.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.
* **Selected Item Foreground Color** – The foreground color of the widget when selected.
  + **Red**
  + **Green**
  + **Blue**
* **Selected Item Background Color** – The background color of the widget when selected.
  + **Red**
  + **Green**
  + **Blue**

## Dynamic Text Label widget

This widget can display a dynamic text from a signal source. It can be connected to a function block that out puts a string that for example can consist of a numeric value with a prefix, e.g “100km/h”.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Right Aligned** – Defines if the text should be right aligned with in its area.
* **Font** – The font used to display the string.
* **Signal Sink** – The signal source connected to.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Dynamic Value widget

This widget can display a dynamic numeric value from a signal source. It can for example be used to show a digital speedometer.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Right Aligned** – Defines if the text should be right aligned with in its area.
* **Font** – The font used to display the string.
* **Signal Sink** – The signal source connected to.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Dynamic Label widget

This widget can display any label based on a label ID from a signal source. It can be used for constructing a menu for example.

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes.

* **Name**
* **X** – The x position based on the area it’s placed in.
* **Y** – The y position based on the area it’s placed in.
* **Width** – The width of the area to display the images in.
* **Height** – The height of the area to display the images in.
* **Boxed** – Defines if the widget should have a border or not.
* **LabelToDisplay** – The input signal that needs to be connected to a source.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

## Dynamic Image Widget

This widget can display different images based on the image ID from a signal source. It can be used to build the menu for example or display warning images.

It allocates one layer on hardware rendered platforms.

If an animation is selected the animation properties will be visible. Depending on the settings for the animation in the Animation Editor, some of the properties here might be disabled.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it’s placed in.
* **Y** – The y position based on the area it’s placed in.
* **Width** – The width of the area to display the images in.
* **Height** – The height of the area to display the images in.
* **Boxed** – Defines if the widget should have a border or not.
* **ImageIDToDisplay** – The input signal that needs to be connected to a source
* **Animation1** – First animation slot.
  + **Use Triggers 1** – Select if triggers should be used to start the animation.
    - **Behaviour on Load** – Specifies the behavior of the animation when the layout is loaded.
      * **Do Nothing** – Does nothing
      * **Run One Cycle** – Runs on cycle
      * **Show Default Frame** – Shows the default frame set for the animation in the Animation Editor.
    - **Behaviour on Trigger Active** – Specifies the behavior of the animation when the trigger goes to active.
      * **Start from Beginning** – Specifies if the animation should start from the beginning or if it should start from where it’s at.
    - **Behaviour on Trigger Inactive** – Specifies the behavior of the animation when the trigger goes to inactive.
      * **Stop** – Stops immediately when the trigger is inactive
      * **Stop at end of cycle** – Stops when the animation reaches the end of the cycle.
      * **Stop at Default Frame** – Stops at the default frame specified for the animation in the Animation Editor.
      * **Continue** – The animation continue to run until the Layout changes.
* **Animation2** – Second animation slot.
  + **Use Triggers 2** – Select if triggers should be used to start the animation.
    - **See the settings under Use Triggers 1**

***Note****: Make sure that the images that should be displayed by the dynamic image widget has the same size as the widget otherwise the images will be displayed incorrectly.*

## Selectable List widget

This widget can be used in the menu to make settings in the system, for example set language. It can also be used for showing a number of messages in a list, for example diagnosis error messages (DTC).

It does not allocate any layer on hardware rendered platforms.

The widget has the following attributes:

* **Name**
* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width** – The maximum width for the widget. If the text in the label is wider than the width then the text will be cut after the last character that fitted.
* **Height** – The maximum height of the widget.
* **Boxed** – Defines if the widget should have a border or not.
* **Selectable** – Defines if the widget should be used to set a value from a list or just to show a number of labels in a list.
* **CurrentValue** – The signal source that the current active settings is read from. This is not used if Selectable isn’t checked.
* **LabelToValueMappings** – The signal source of the container that holds the different available settings for the widget.
* **SelectedValue** – An output signal sent to a receiving function block to store the new selected value. This is not used if Selectable isn’t checked.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.
* **Selected Item Foreground Color** – The foreground color of the widget when selected. Has no effect if the Selectable property isn’t set.
  + **Red**
  + **Green**
  + **Blue**
* **Selected Item Background Color** – The background color of the widget when selected. Has no effect if the Selectable property isn’t set.
  + **Red**
  + **Green**
  + **Blue**

## Signal Trigger widget

This widget can be used to send a signal from a menu item. This can be used to reset the trip distance for example.

It does not allocate any layer on hardware rendered platforms.

It has the following attributes:

* **Name**
* **X** – Has no real effect other that it can be used to display in Preview window which menu item the signal trigger widget is connected to.
* **Y** – Has no real effect other that it can be used to display in Preview window which menu item the signal trigger widget is connected to.
* **Width** – Has no effect.
* **Height** – Has no effect.
* **Boxed** – Do not use. Will create a border if selected.
* **Signal Trigger Source** – Do not use. Due to a bug it will show different available inputs depending on which dynamic label was shown last but it should not be changed.

## List Navigation Widget

This widget is used for navigation in the menu by creating a list representation of the menu. It requires that a menu has been added to the project.

It does not allocate any layer on hardware rendered platforms.

It has the following attributes:

* **X** – The x position based on the area it is placed in.
* **Y** – The y position based on the area it is placed in.
* **Width –** The maximum width of the widget
* **Height –** The maximum height of the widget
* **Boxed** – Defines if the widget should have a border or not.
* **Display Item Count –** Defines how many lines the menu should consist of. The widget will automatically adjust the size of each menu item to fit to the layout area and add borders to each item as well.
* **Foreground Color**
  + **Red** – The red part of a RGB value describing the foreground color.
  + **Green** – The green part of a RGB value describing the foreground color.
  + **Blue** – The blue part of a RGB value describing the foreground color.
* **Background Color**
  + **Red** – The red part of a RGB value describing the background color.
  + **Green** – The green part of a RGB value describing the background color.
  + **Blue** – The blue part of a RGB value describing the background color.

# Animations

Animations are used by objects that allocate a layer\* and can hence only be used on hardware rendered platforms. The animations use the same layer as the object that that it’s tied to, i.e. no additional layer is used by an animation. Each widget/area can have maximum two animations tied to them.

\* ImageWidget, DynamicImageWidget.

To define/create an animation, navigate to the “Animation Editor” tab. Each animation type have a sub tab where the specific animations are defined. Each sub tab have a list (to the left on the screen) of all the animations (of the specific type) that are currently defined. This is also where animations are created and removed.

To the right of the animation list each sub tab have an area which is used to; add/select/prioritize resources and/or define/preview the properties of an animation. The look of this area differs depending on which animation type is currently being created/edited.

## Frame-by-Frame animation

This animation type uses several images to create the illusion that the image object is moving/changing color or similar effect. The animation type can **only** be used by the DynamicImageWidget.

It has the following attributes:

* **Name**
* **Frame Duration** – How long each frame (image) shall be displayed (in milliseconds)
* **Repeat Mode** – Defines if the animation shall be; played once, looped, bounced (played backwards when last animation step have been reached), or bounced loop.
* **Default Frame** – Defines the default frame that can be used in the widget properties when settings up the widget in the Layout Editor.
* **CurrentImageID** – The source signal for the DynamicImageWidget that is tied to the animation.

## Opacity animation

This animation type is used to “fade in“/“fade out” objects.

It has the following attributes:

* **Name**
* **Length** – The total time of the animation (in milliseconds).
* **Repeat Mode** – Defines if the animation shall be; played once, looped, bounced (played backwards when last animation step have been reached), or bounced loop.
* **Start opacity** – The layer’s opacity when the animation start (in percent).
* **Stop opacity** – The layer’s opacity when the animation stop (in percent).
* **Default Opacity** – Sets the default opacity that can be used when setting up the properties for the widget using the animation in Layout Editor.
* **Curve type** – Defines the “change speed” and the opacity (in percent) during the animation length.

## Translation animation

This animation type is used to move objects along a path. The path types supported are; linear and Bezier curve.

It has the following attributes:

* **Name**
* **Length** – The total time of the animation (in milliseconds).
* **Repeat Mode** – Defines if the animation shall be; played once, looped, bounced (played backwards when last animation step have been reached), or bounced loop.
* **Default Position** – Defines the default frame that can be used in the widget properties when settings up the widget in the Layout Editor.
* **Path** – Defines the type of path (Linear or Bezier Curve) that the object shall move along. (The actual path is manipulated in the preview window of
* **Relative start X** – The animation’s start position along the X-axis (in pixels) relative to the layer’s (display area, widget’s) final position (as defined in the layout editor).
* **Relative start Y** – The animation’s start position along the Y-axis (in pixels) relative to the layer’s (display area, widget’s) final position (as defined in the layout editor).
* **Bezier C0 X** – Weight point for the animation start position along the X-axis.
* **Bezier C0 Y** – Weight point for the animation start position along the Y-axis.
* **Bezier C1 X** – Weight point for the animation stop position along the X-axis.
* **Bezier C1 Y** – Weight point for the animation stop position along the Y-axis.

**Easing Curve Type** – Defines the “change speed” and the position of the object during the animation length.