TASKING®

# Using the TASKING Software Platform for ARM

MA101-869 (v1.3) July 29, 2015

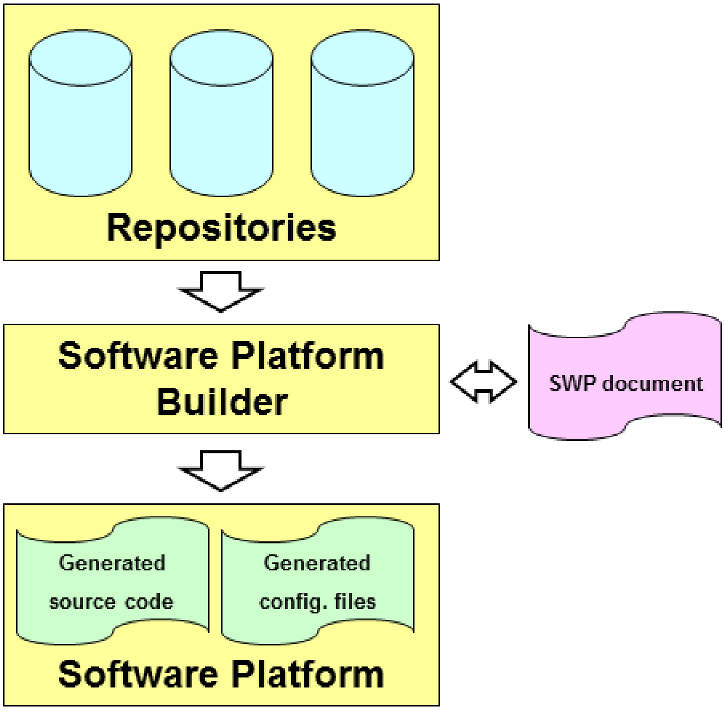
# Table of Contents

# Chapter 1. Introduction

With TASKING Software Platform you can quickly create full-featured applications. A Software Platform is made up of software blocks, pieces of functionality that you can use in your application, like RTOS facilities, peripheral access or software protocols. The exact contents of a Software Platform depend on the need of your application.

The Software Platform Builder is used to manage your Software Platform. It is both a graphical editor and a code generator. Collections of software modules are delivered as Software Platform repositories.

The following figure shows the process to create your own Software Platform:



A Software Platform repository may contain any kind of software, but typical modules include interrupt services, timers, peripherals (hardware wrappers), drivers, kernel services (such as POSIX multithreading), device I/O, file system (FatFs), networking (TCP/IP), graphical user interface, etc.

The main reasons for having software modules are:

- Integration. Modules do not live isolated. Quite the contrary, they need to relate and interact with other modules. For instance, they often define interfaces which are implemented by other modules.

- Software reuse. Software reuse means effectively to be capable of encapsulating all the information of a component (hardware and/or software) in a consistent manner. Once this information is made available, higher abstraction layers can use it to customize or configure the system after little (or none) user intervention.

- Hide complexity. Writing software from scratch is a time-consuming activity. To write software that accesses or controls a peripheral, you need a thorough knowledge of how the peripheral works: which registers you need, which device specific commands to use, which communication protocols to use, and which interrupts to handle.

  The modules of the Software Platform repository take care of all these laborious lower level routines and instead provide you with an easy to use Application Programming Interface (API) for each peripheral that your application needs to control.

- Configurability. Embedded systems require very precise (compile-time) configuration mechanisms to manage their intrinsic complexity. A typical module may show different possible configurations that have direct impact on the final behavior of the software. The main idea behind compile-time configurations in embedded systems is to remove unnecessary functionality in order to enhance memory costs and real-time performance.

- Expandability. Altium, but also other parties, can adapt the modules and create their own modules.

For information how to use the Software Platform Builder, see Chapter 3, *Getting Started with the Software Platform Builder*.

# Chapter 2. Organization of the Software Platform Repositories

Software Platform repositories can contain numerous software modules that take care of lower level software routines as well as modules that offer extra functionality by providing you with a convenient API.

The Software Platform consists of device stacks and software services. This chapter describes both parts and how they are related.

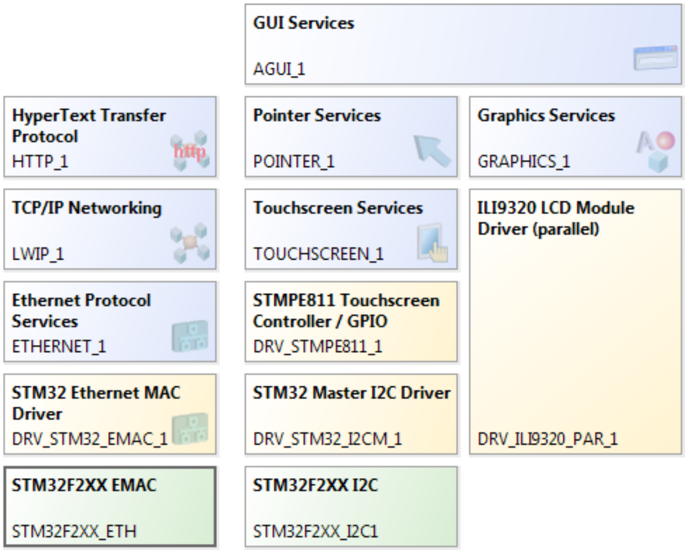
## 2.1. Device Stacks

Device stacks are all about making hardware peripherals available to application code through abstract and generic software interfaces. By placing more or less modules on a stack, you can choose the abstraction level you want to use in your application. The lowest level modules are specific for a particular hardware device. On top of that, you can stack higher level modules that provide more generic functionality to access the device. For example, at the higher, abstract level, you could choose to use a module to access a file system in your application. At the lower levels you still can select modules to decide which specific storage device you want to access (a hard drive, SD card, RAM drive, ...) Thus, the lower level modules are more specific for a particular peripheral while the higher level modules are less hardware specific and can even be used in combination with multiple peripheral devices.

You can use the Software Platform Builder editor to build device stacks.

### Example

The following figure contains a HTTP device stack and a GUI device stack.

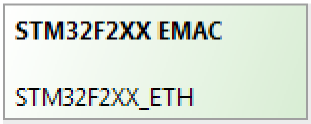


Each colored stack item represents a software module. In this example the HTTP stack consists of five modules, for each abstraction level one. The GUI Services requires Graphics Services and Pointer Services. In general, your application interfaces only to the highest level modules.

Device stacks may be composed of the following types of plug-ins: peripherals, drivers, stack services and adaptors.

### 2.1.1. Peripherals

Peripherals (the green stack items) are the lowest level modules. They provide information for the higher layers of the stack to access the peripherals. Information such as the base address, interrupt assignment and any soft peripheral configuration is all stored in the peripheral.



**STM32F2XX EMAC**

STM32F2XX_ETH

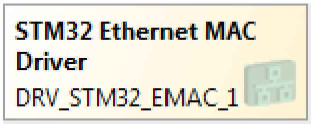
Peripheral modules are meant to be instantiated. Each instance then corresponds to one device of the given kind. Instances of peripherals do not normally require other instances.

In most situations, your application does not access the peripherals directly, because the application accesses them through the driver's interface on top of it.

For most peripherals you need to use the TASKING Pin Mapper to configure the connections between peripherals and port pins. For information about the TASKING Pin Mapper see *Using the TASKING Pin Mapper for ARM*.

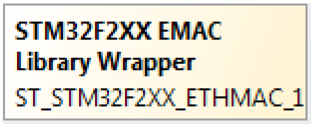
### 2.1.2. Drivers

Drivers (the yellow stack items) provide the next level of abstraction. They provide low-level access to specific hardware either via a peripheral or via another driver.



**STM32 Ethernet MAC Driver**

DRV_STM32_EMAC_1



**STM32F2XX EMAC Library Wrapper**

ST_STM32F2XX_ETHMAC_1

The difference between a driver and a peripheral is well defined. The peripheral only defines basic information about the hardware but provides no further functionality. This basic information can be used by a driver which you can place on top of the peripheral.

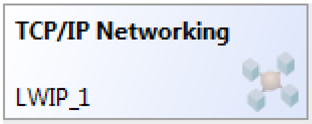
Driver modules are meant to be instantiated. Each instance then corresponds to one device of the given kind. A driver instance requires one instance of the corresponding peripheral.

Drivers are hardware specific and so are the interfaces they offer. Hence, if your application contains code that accesses a driver's API, your application will be hardware dependent. Drivers still operate at a low abstraction level, and using a device at the driver level requires knowledge of that particular driver's interface.

Some drivers are available as library wrappers. They only offer an initialization routine which you can call from an application in order to configure pins, clocks and registers. After this initialization you can use the peripheral driver libraries from silicon vendors.

## 2.1.3. Stack Services

Stack services (the blue stack items) are the most abstract layers of functionality.



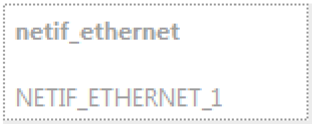
TCP/IP Networking

LWIP_1

Stack services provide standardized, hardware independent access to devices. Stack services are instantiated and ensure portability of your application. For example, if you run an application with a generic TCP/IP service, the application remains the same if you replace Ethernet with PPP and serial. A portable TCP/IP application works exactly the same, whether linked to a UART or a to an EMAC peripheral, since the Software Platform handles the lower level details.

## 2.1.4. Adaptors

Adaptors (the grayed out stack items) provide a mechanism to link higher level services to hardware specific modules.



netif_ethernet

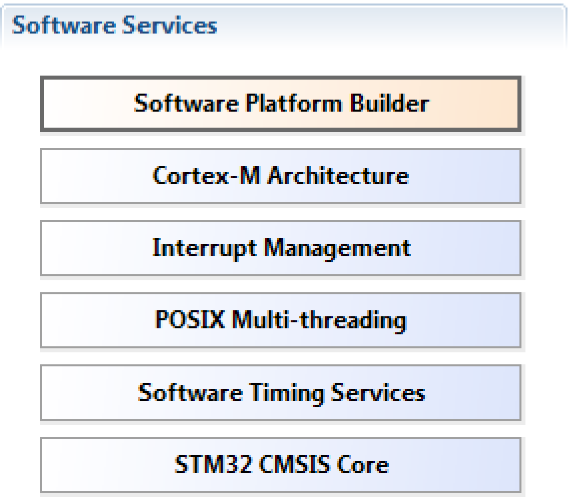
NETIF_ETHERNET_1

Adaptors do not have options you can change. By default, adaptors are not visible in the device stacks.

You can make adaptors visible in the Preferences dialog. See Section 3.7, *Software Platform Builder Preferences*.

## 2.2. Software Services

Some services are static and not meant to be instantiated. They have no (direct) relationship with peripherals or other services. They facilitate common functionality, like POSIX multithreading support. They are shown in your editor at the left side of the **Device Stacks**. Those services may also be added automatically if required by other services.



# Chapter 3. Getting Started with the Software Platform Builder

The Software Platform Builder is the graphical interface to manage the plug-ins of the Software Platform repositories. The Software Platform Builder enables you to:

- add peripherals
- build device stacks
- add software services
- configure all software services and stack items
- generate code for your Software Platform

## 3.1. Creating a Software Platform Document

To use the Software Platform Builder you need a project. The Software Platform Builder becomes available when you add a special Software Platform document to your project.

### Add a Software Platform document to an existing project

1. First make sure you have an existing project. This is explained in the *Getting Started* manual of the toolset. In this example we assume you have a project called `myproject`.

2. From the **File** menu, select **New » TASKING Software Platform Document**.

   *The New Software Platform Document wizard appears.*

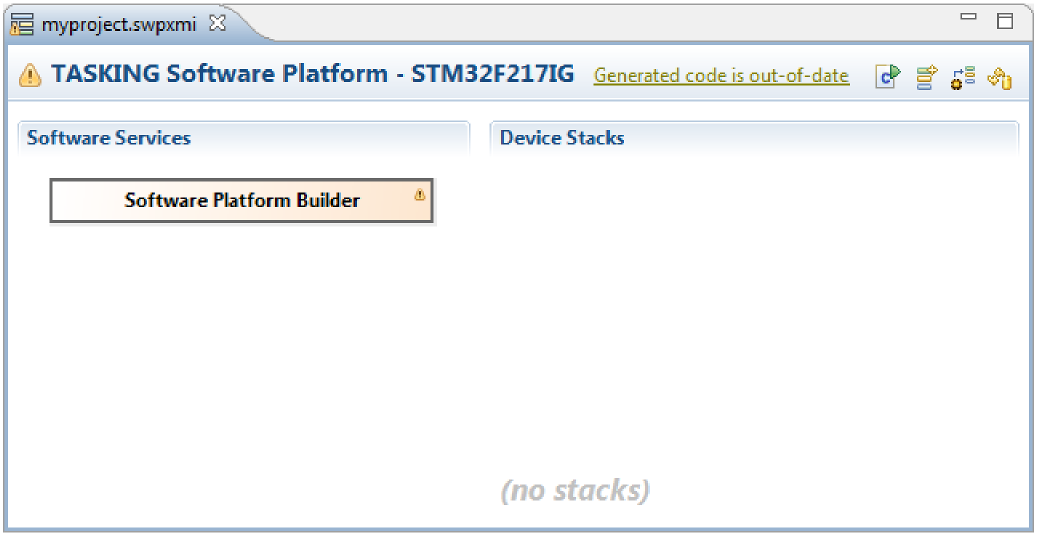
3. Select the **Project** folder for the Software Platform document: type the name of your project (`myproject`) or click the **Browse** button to select a project.

4. In the **File name** field, enter a name for the Software Platform document, for example `myproject.swpxmi` and click **Finish**.

   *A new Software Platform document is added to the project with extension `.swpxmi`.*

### TASKING Software Platform Builder editor

Double-click on the Software Platform document to open it. The editor consists of two sections: the **Software Services** and the **Device Stacks**.


myproject.swpxmi
TASKING Software Platform - STM32F217IG
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
(no stacks)

The following toolbar icons are available:

| Icon | Action | Description |
|---|---|---|
| | Generate Code | Generates the source code and adds it to your project. |
| | Add | Adds software services and/or device stacks. |
| | Synchronize with Pin Mapper output | Imports Pin Mapper settings. If you already have configured peripherals in the Pin Mapper, the settings are synchronized with the Software Platform. |
| | Reload Repository | Reloads the repository. This can be necessary when you have updated the repository or when you changed the contents of a local repository. |

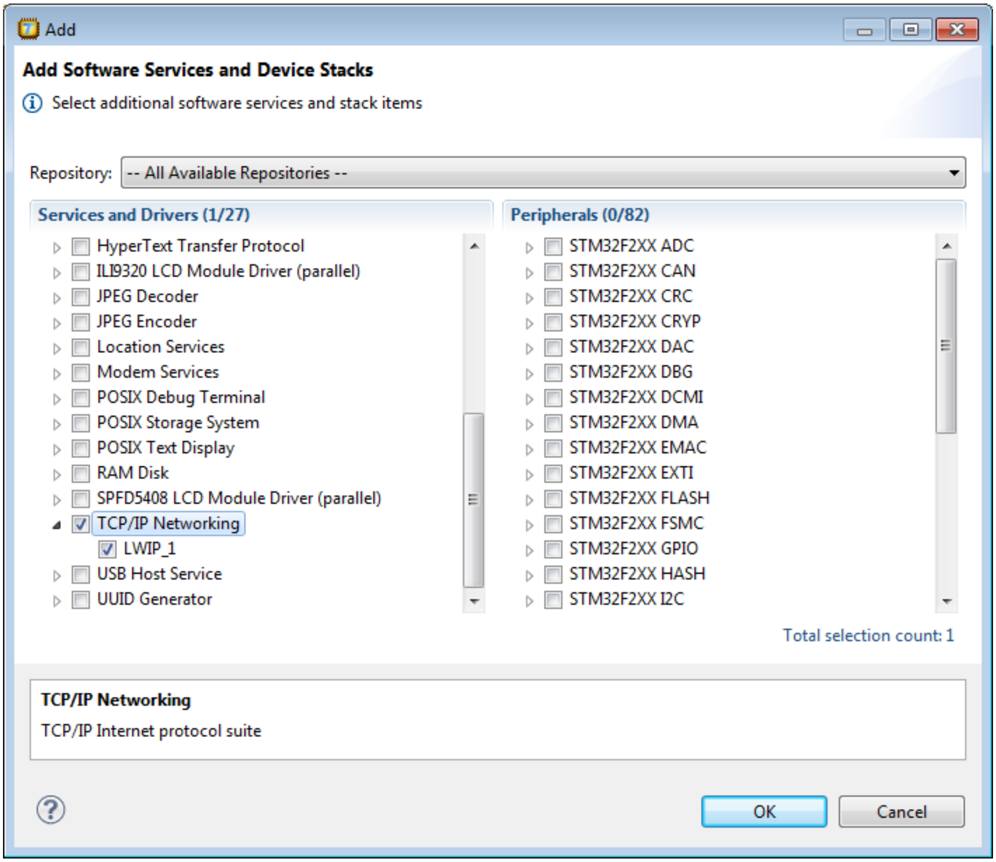
## 3.2. Working with Device Stacks

In the **Device Stacks** section you can build device stacks by adding modules from one or more Software Platform repositories. To build a device stack, you can start bottom-up by choosing a peripheral or you can start top-down by choosing a high-level stack service.

### Add a software service or device stack

1. From the **Software Platform** menu, select **Add...** or click .

   *The Add Software Services and Device Stacks dialog appears. The left side shows the stack services, software services and drivers. The right side shows all peripherals for the selected device. You can filter on a specific repository. By default all available repositories are listed.*



Add

**Add Software Services and Device Stacks**

Select additional software services and stack items

Repository: -- All Available Repositories --

Services and Drivers (1/27)

- □ HyperText Transfer Protocol
- □ ILI9320 LCD Module Driver (parallel)
- □ JPEG Decoder
- □ JPEG Encoder
- □ Location Services
- □ Modem Services
- □ POSIX Debug Terminal
- □ POSIX Storage System
- □ POSIX Text Display
- □ RAM Disk
- □ SPFD5408 LCD Module Driver (parallel)
- ☑ TCP/IP Networking
  - ☑ LWIP_1
- □ USB Host Service
- □ UUID Generator

Peripherals (0/82)

- □ STM32F2XX ADC
- □ STM32F2XX CAN
- □ STM32F2XX CRC
- □ STM32F2XX CRYP
- □ STM32F2XX DAC
- □ STM32F2XX DBG
- □ STM32F2XX DCMI
- □ STM32F2XX DMA
- □ STM32F2XX EMAC
- □ STM32F2XX EXTI
- □ STM32F2XX FLASH
- □ STM32F2XX FSMC
- □ STM32F2XX GPIO
- □ STM32F2XX HASH
- □ STM32F2XX I2C

Total selection count: 1

**TCP/IP Networking**

TCP/IP Internet protocol suite

2. (Optional) Select a **Repository**.

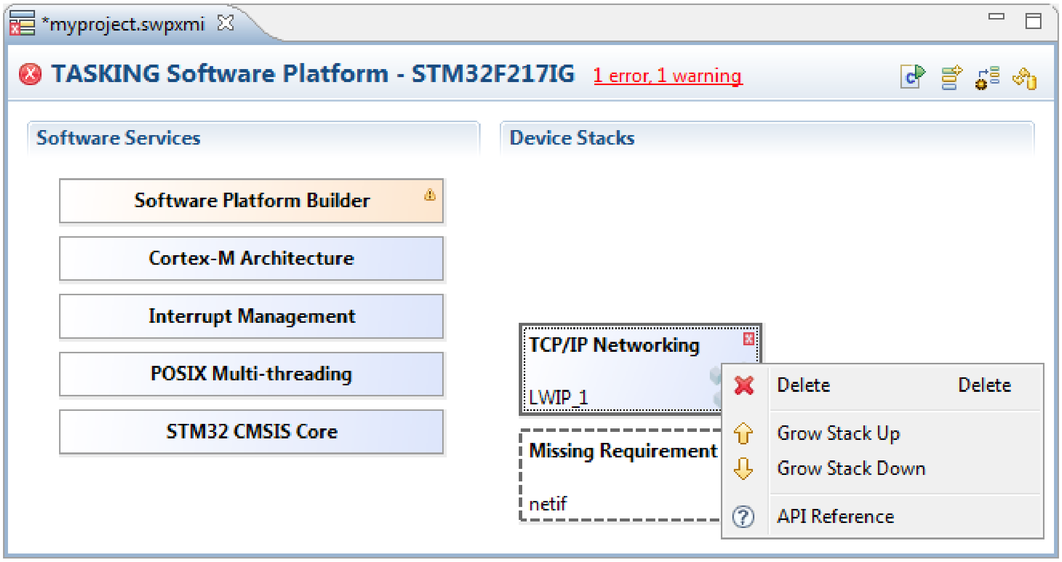
3. Select the services, drivers and peripherals you want and click **OK**.

   *The new device stacks and software services appear in the TASKING Software Platform editor. Some software services appear automatically because they are required by other services or stack items.*

If you do not see any peripheral, you probably selected an unsupported device or a generic architecture.

## Growing stacks

Once you have added one or more stack items, you can use **Grow Stack Up** or **Grow Stack Down** from the context menu to complete your device stack.



The **Grow Stack Up** and **Grow Stack Down** menu items only become available if possibilities exist to extend the stack from the point of the selected stack item. Once you have partial device stacks on the document, you can extend them with other stack items to grow them to the abstraction level you wish to use in your application. For an interface description of a module select **API Reference**.

To grow a stack up:

1. Right-click on a plug-in of the existing stack and select **Grow Stack Up**.

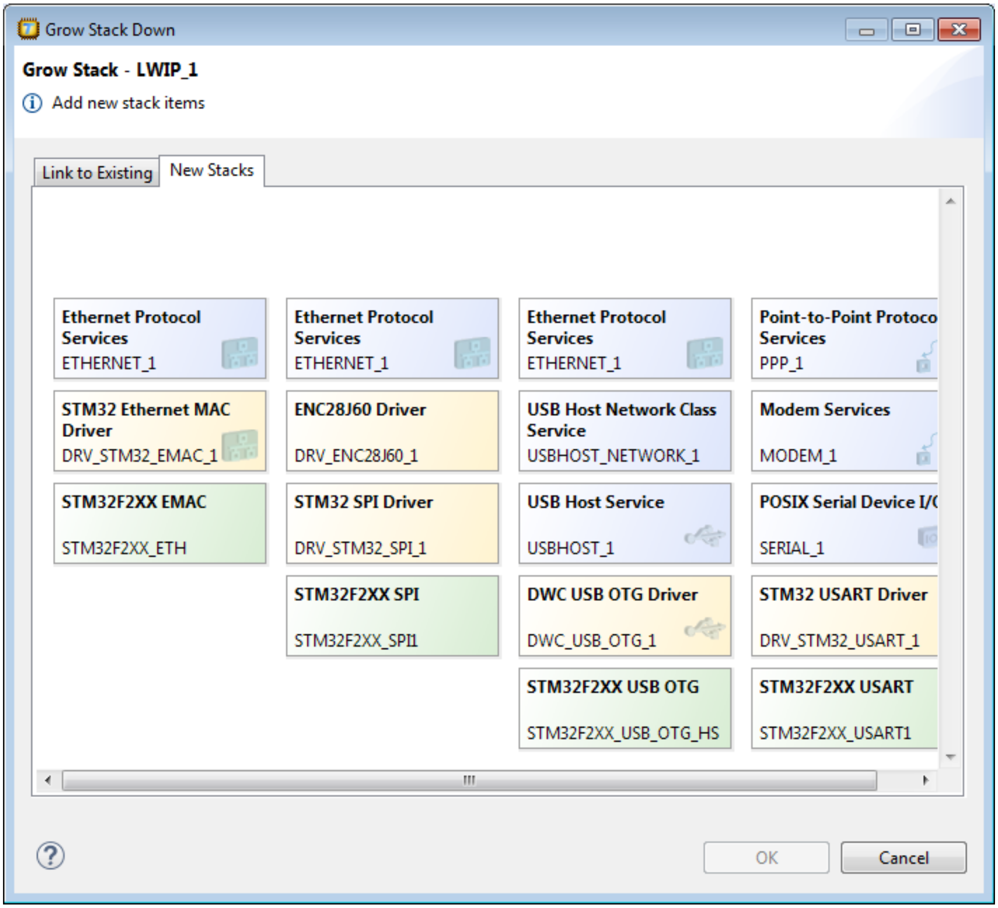
   *The Grow Stack dialog appears. The dialog shows all possible combinations of sub-stacks that you can place on top of the selected module.*

2. Select any module in any of the possible sub-stacks to extend the selected stack upwards and click **OK**.

To grow a stack down:

1. Right-click on a plug-in of the existing stack and select **Grow Stack Down**.

   *The Grow Stack dialog appears. The dialog shows all possible combinations of sub-stacks that you can place below the selected module.*



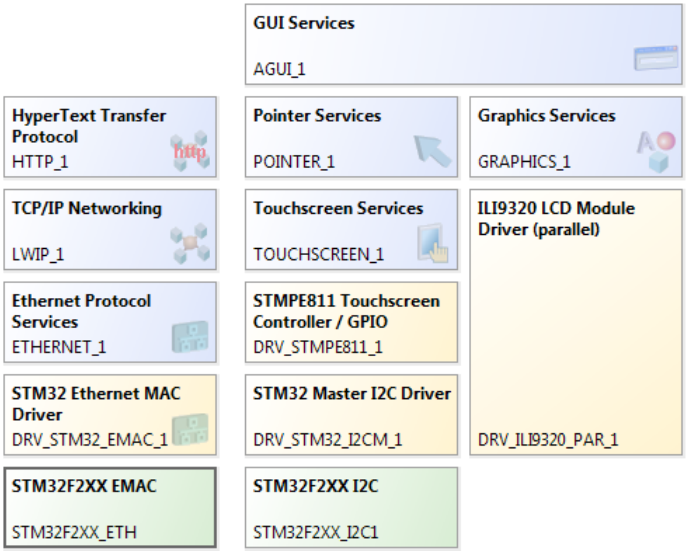
2. Select any module in any of the possible sub-stacks to extend the selected stack downwards and click **OK**.

Note that when you click on a **Missing Requirement**, the Grow Stack dialog appears automatically.

The Grow Stack dialog contains two tabs, one to form a new stack and one to link two existing stacks. If your document contains partial stacks which potentially can be combined into a single stack, it will be visible in the **Link to Existing** tab. You can link them together to form a complete stack. Otherwise select the **New Stacks** tab.

## Example

The following figure shows some device stacks.



GUI Services
AGUI_1

HyperText Transfer Protocol
HTTP_1

Pointer Services
POINTER_1

Graphics Services
GRAPHICS_1

TCP/IP Networking
LWIP_1

Touchscreen Services
TOUCHSCREEN_1

ILI9320 LCD Module Driver (parallel)
DRV_ILI9320_PAR_1

Ethernet Protocol Services
ETHERNET_1

STMPE811 Touchscreen Controller / GPIO
DRV_STMPE811_1

STM32 Ethernet MAC Driver
DRV_STM32_EMAC_1

STM32 Master I2C Driver
DRV_STM32_I2CM_1

STM32F2XX EMAC
STM32F2XX_ETH

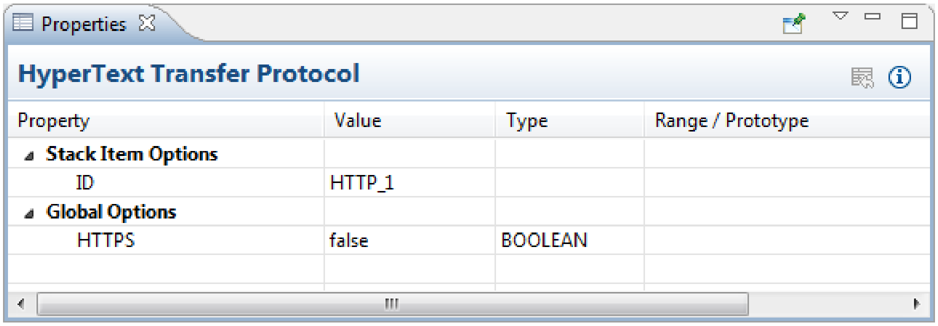
STM32F2XX I2C
STM32F2XX_I2C1

Note that some drivers (ILI9320 LCD Module Driver in this example) do not require a peripheral, because the device is accessed through the memory controller.

## 3.3. Configuring the Software Platform Plug-ins

You can configure peripherals, drivers, stack services and software services. Stack items can have individual options and/or global options. Individual options are only valid for a specific stack item, global options are valid for all stack items of the same type. When you click on an item in the Software Platform Builder, configuration options appear in the Properties view.

For example, when you click **HyperText Transfer Protocol**, a view similar to the following appears:



HyperText Transfer Protocol

| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| **Stack Item Options** | | | |
| ID | HTTP_1 | | |
| **Global Options** | | | |
| HTTPS | false | BOOLEAN | |

A plug-in option may automatically set other plug-in options and may even add automatic services. For example, in this case when you set HTTPS to `true` this adds the Matrix SSL / TLS services automatically.

## Connecting peripherals to pins

For most peripherals you need to specify which pins are used. You can do this with the TASKING Pin Mapper.

1. Add a TASKING Pin Mapper Document (**File » New » TASKING Pin Mapper Document**) and configure the pins as explained in *Using the TASKING Pin Mapper for ARM*.

   *This results in a file called `myproject.pincfg`.*

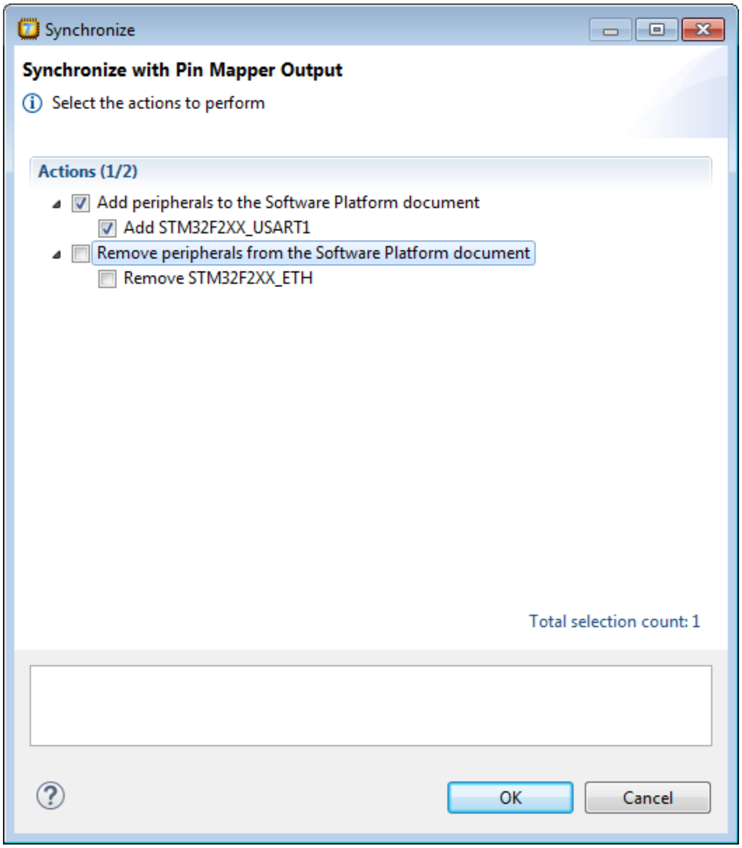
2. From the **Pin Mapper** menu, select **Generate Code** or click .

   *The Pin Mapper sources are generated and are added to your project in the folder `PinMapper`.*

3. Go back to the TASKING Software Platform document.

4. From the **Software Platform** menu, select **Synchronize with Pin Mapper Output** or click .

   *The Synchronize dialog appears.*



Synchronize

**Synchronize with Pin Mapper Output**

Select the actions to perform

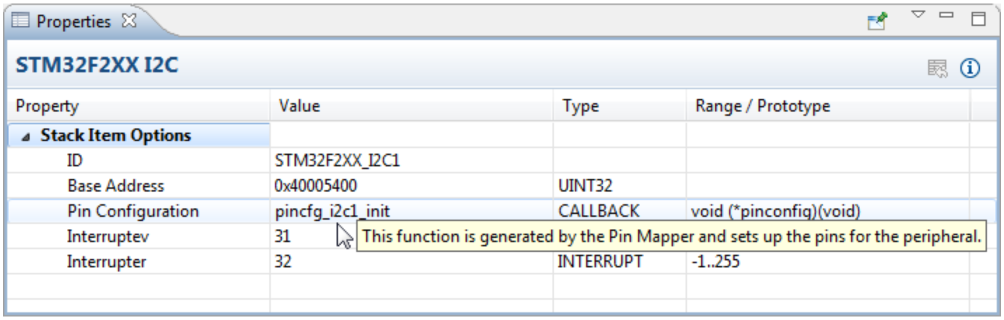
Actions (1/2)

- ☑ Add peripherals to the Software Platform document
  - ☑ Add STM32F2XX_USART1
- □ Remove peripherals from the Software Platform document
  - □ Remove STM32F2XX_ETH

Total selection count: 1

5. Select the peripherals you want to add to or remove from the Software Platform document and click **OK**.

The Software Platform uses the output of the TASKING Pin Mapper to configure the peripherals. For example, click on the STM32F2XX I2C peripheral to show its properties.



STM32F2XX I2C

| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| Stack Item Options | | | |
| ID | STM32F2XX_I2C1 | | |
| Base Address | 0x40005400 | UINT32 | |
| Pin Configuration | pincfg_i2c1_init | CALLBACK | void (*pinconfig)(void) |
| Interruptev | 31 | | |
| Interrupter | 32 | INTERRUPT | -1..255 |

This function is generated by the Pin Mapper and sets up the pins for the peripheral.

The Software Platform uses the function `pincfg_i2c1_init` to configure the STM32F2XX I2C peripheral. This function has been generated by the TASKING Pin Mapper.

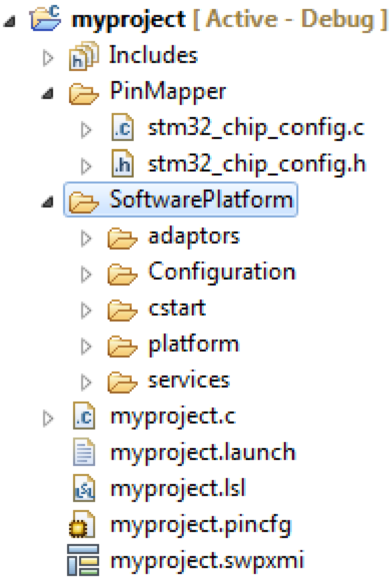
# 3.4. Generating and Using the Source Code

When you have added, configured and/or updated the Software Platform plug-ins you are ready to add the Software Platform sources to your project.

## Generate source code

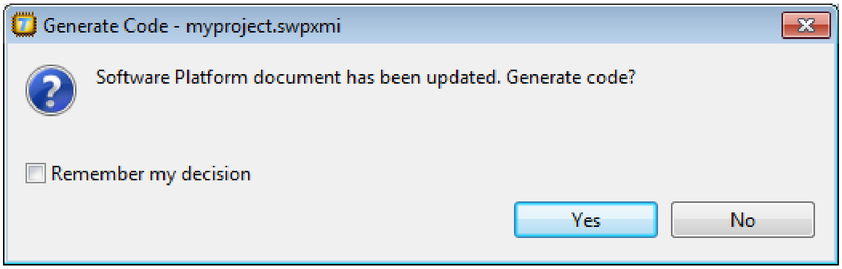
- From the **Software Platform** menu, select **Generate Code** or click .

  *The Software Platform sources are retrieved from the repositories and are added to your project in the folder `SoftwarePlatform`. Tool options such as defines and include paths are added to the Generated options field in your project properties. Options can also be set on the `SoftwarePlatform` folder and/or sub-folders.*



- myproject [ Active - Debug ]
  - Includes
  - PinMapper
    - stm32_chip_config.c
    - stm32_chip_config.h
  - SoftwarePlatform
    - adaptors
    - Configuration
    - cstart
    - platform
    - services
  - myproject.c
  - myproject.launch
  - myproject.lsl
  - myproject.pincfg
  - myproject.swpxmi

Also, every time you save the changes you have made to your Software Platform document, a dialog appears asking if you want to generate the code. You can change this behavior in the Preferences dialog as explained in Section 3.7, *Software Platform Builder Preferences*.



Generate Code - myproject.swpxmi

Software Platform document has been updated. Generate code?

□ Remember my decision

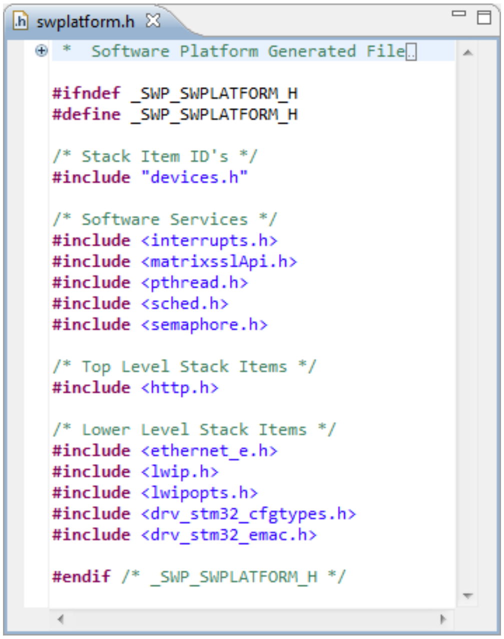
## Using the Software Platform sources

A plug-in defines a set of functionality - that is, types, defines, structures and functions. This functionality is accessible from your application. You can find the generated files in the `SoftwarePlatform` folder of your project. To use the generated files, you must add a `#include` statement into your top-level source file (typically `main.c`):

```
#include "swplatform.h"
```

All other required header files needed to access the devices in the stack are included (made available) as a result of including this file.

The following is an example of `swplatform.h` generated as a 'wrapper' around the active project's Software Platform:



```
* Software Platform Generated File

#ifndef _SWP_SWPLATFORM_H
#define _SWP_SWPLATFORM_H

/* Stack Item ID's */
#include "devices.h"

/* Software Services */
#include <interrupts.h>
#include <matrixsslApi.h>
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>

/* Top Level Stack Items */
#include <http.h>

/* Lower Level Stack Items */
#include <ethernet_e.h>
#include <lwip.h>
#include <lwipopts.h>
#include <drv_stm32_cfgtypes.h>
#include <drv_stm32_emac.h>

#endif /* _SWP_SWPLATFORM_H */
```

You can now use the functions of the Software Platform in your sources. For example,

```
int main( void )
{

    http_t *http;

    http = http_open(HTTP_1);
    if (!http)
    {
        fprintf(stderr,"HTTP open failed\n");
        exit(1);
    }
    ...
}
```

## 3.5. Using the Sample Projects

A Software Platform repository may contain a number of sample projects. These examples can be a good starting point for your own project.

You can import Software Platform repository examples via the Import wizard in Eclipse.

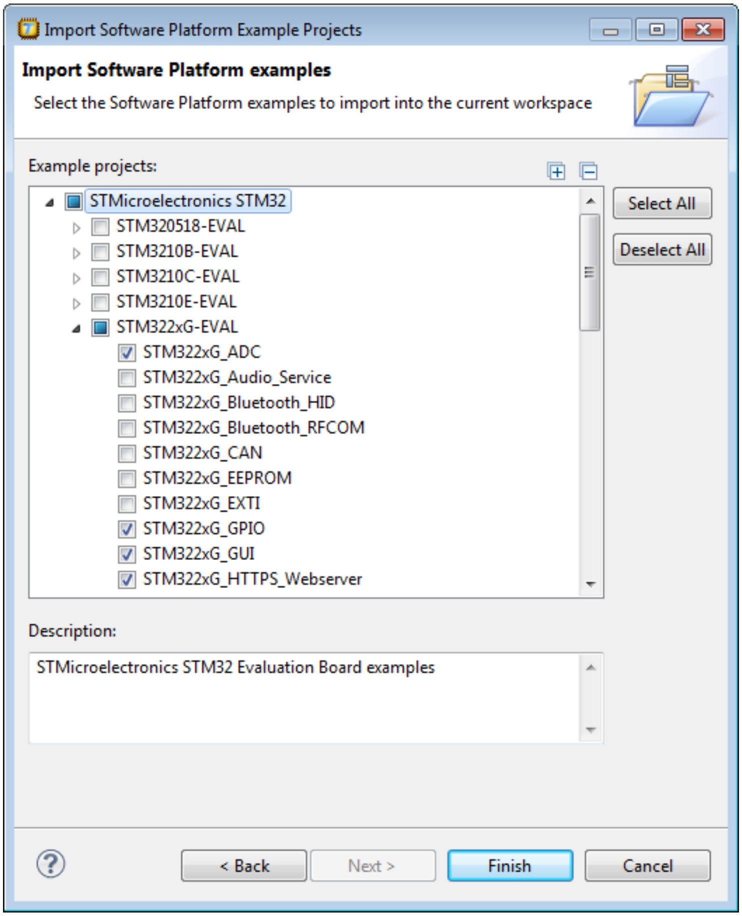
## Import an existing Software Platform project

1. From the **File** menu, select **Import**.

   *The Import dialog appears.*

2. Select **TASKING Software Platform » Example Projects** and click **Next**.

   *The Import Software Platform Examples dialog appears.*



Import Software Platform Example Projects

**Import Software Platform examples**

Select the Software Platform examples to import into the current workspace

Example projects:

- ■ STMicroelectronics STM32
  - □ STM320518-EVAL
  - □ STM3210B-EVAL
  - □ STM3210C-EVAL
  - □ STM3210E-EVAL
  - ■ STM322xG-EVAL
    - ☑ STM322xG_ADC
    - □ STM322xG_Audio_Service
    - □ STM322xG_Bluetooth_HID
    - □ STM322xG_Bluetooth_RFCOM
    - □ STM322xG_CAN
    - □ STM322xG_EEPROM
    - □ STM322xG_EXTI
    - ☑ STM322xG_GPIO
    - ☑ STM322xG_GUI
    - ☑ STM322xG_HTTPS_Webserver

Description:

STMicroelectronics STM32 Evaluation Board examples

3. Select the Software Platform examples you want to import into the current workspace and click **Finish**.

   *The original examples are copied into the current workspace and the Software Platform sources and Pin Mapper sources are generated.*

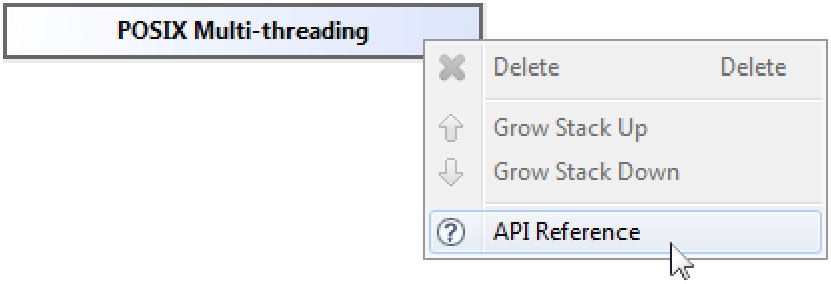
Once the examples are in your workspace you can build the projects.

## 3.6. API Reference

Each repository contains detailed descriptions of its contents. You can access this documentation from the Help menu (**Help » Help Contents » Software Platform Repository Reference**).

### Accessing help on individual drivers or services

- Right-click on a stack item or a software service and select **API Reference**.



*Help appears on the individual stack item.*

## 3.7. Software Platform Builder Preferences

You can use the Preferences dialog in Eclipse to specify how the Software Platform Builder should operate.

### To set preferences

1. From the **Window** menu, select **Preferences**.

   *The Preferences dialog appears.*

2. Select **TASKING » Software Platform Builder**.

   *The Software Platform Builder page appears.*

3. Set your preferences and click **OK**.

You can set the following preferences:

### Generate code on save

By default the Software Platform Builders asks if you want to generate code when you save a document (**Prompt**). You can choose to do this automatically (**Always**) or **Never**.

## Software Platform Editor

- **Show adaptors and internal items**. By default, adaptors (see Section 2.1.4, *Adaptors*) are not visible in the device stacks. This option can be useful when you need support from Altium or when you develop your own plug-ins. Normally, you do not need this option.

## Properties View

- **Show resource info**. By default, information about the resource is not visible in the Properties view. You can use the **Show Resource Info** button (ⓘ) in the Properties view to toggle the resource information on or off, or you can use this preference option to turn resource information on by default.

- **Show hidden options**. By default, several plug-in options are not visible in the Properties view. This option can be useful when you need support from Altium or when you develop your own plug-ins. Normally, you do not need this option.

# Chapter 4. Tutorial Creating a File System on a RAM Disk

In this tutorial we show how to create an example Software Platform project. You will learn how to use the software to create a file system on a RAM disk. This example uses the simulator, so no additional hardware is required.

## 4.1. Create a Project

The first step is to create a C project. This is explained in detail in the *Getting Started* manual of the toolset, but for this tutorial we repeat this step.

1. From the **File** menu, select **New » TASKING ARM C/C++ Project**

   *The New C/C++ Project wizard appears.*

2. Enter a name for your project, for example `Generic_FileSystem`.

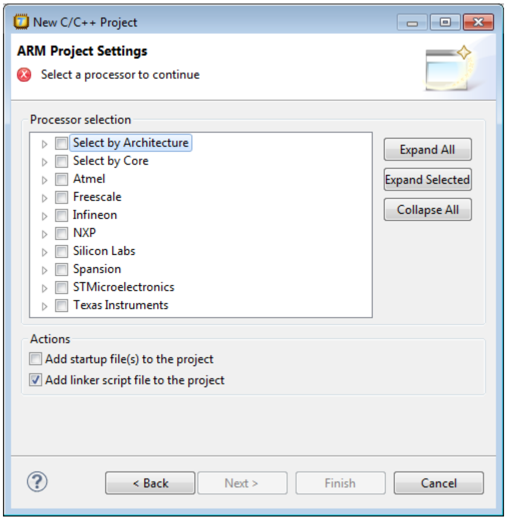
   *In the **Location** field you will see the location where the new project will be stored.*

3. In the **Project type** box you can select whether to create an application or a library.

   - Expand **TASKING ARM Application** and select **Empty Project**. This creates a project without a C source file.

   - Click **Next** to continue.

   *The ARM Project Settings page appears.*



New C/C++ Project

**ARM Project Settings**

Select a processor to continue

Actions

☐ Add startup file(s) to the project

☑ Add linker script file to the project

4. Select the target processor core or architecture for which you want to build the application. For example, architecture ARMv7-M. Afterwards you can always change the processor in the **Project » Properties for** dialog.

5. Leave the rest of the dialog as is and click **Next**.

   *The Target Settings page appears.*

6. In order to debug your project you need to create a debug configuration.

   - Select a target. You can select a target board or a simulator. For this tutorial we select the **ARM Simulator**.

   - Enable **Add launch configuration to the project**. This allows you to debug your project.

   - Leave the other tabs as is. For more information, see section *Creating a Customized Debug Configuration* in Chapter *Using the Debugger* of the *TASKING VX-toolset for ARM User Guide*.

7. Click **Finish** to finish the wizard and to create the project.

   *The project has now been created and is the active project.*

## 4.2. Add a Software Platform Document

Now that we have a project we can add a Software Platform document.

1. From the **File** menu, select **New » TASKING Software Platform Document**.

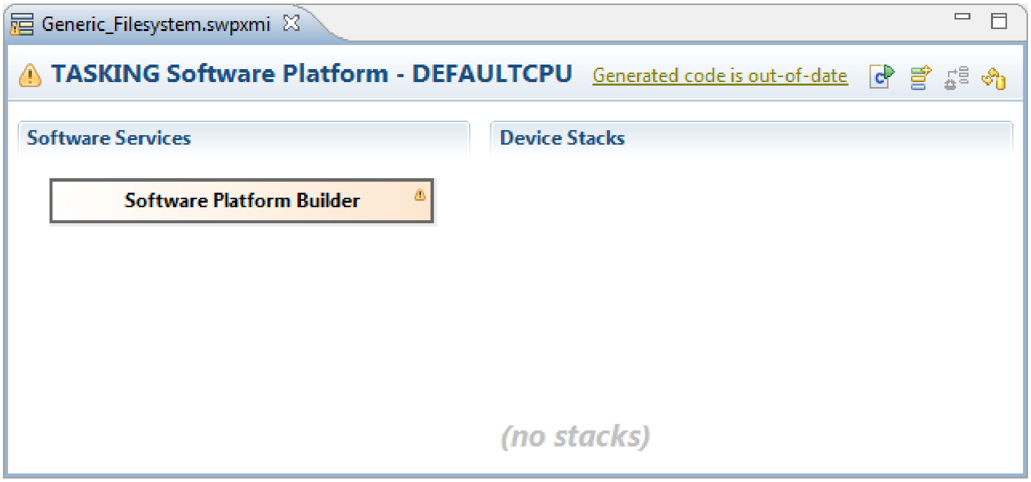
   *The New Software Platform Document dialog appears.*

2. Leave the project name and file name as is and click **Finish**.

   *A new Software Platform document is added to the active project (`Generic_FileSystem.swpxmi`).*

### TASKING Software Platform Builder editor

Double-click on the Software Platform document `Generic_FileSystem.swpxmi` to open it. The editor consists of two sections: the **Software Services** and the **Device Stacks**.


Generic_Filesystem.swpxmi
TASKING Software Platform - DEFAULTCPU
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
(no stacks)

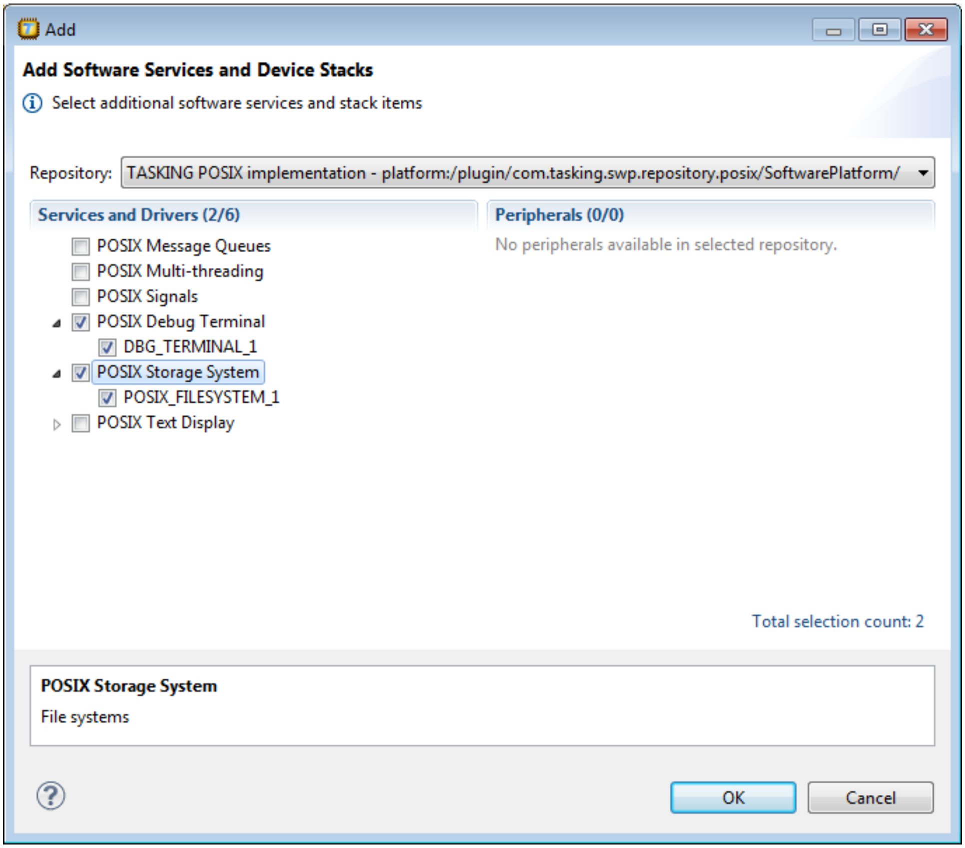
## 4.3. Add the Services and Device Stacks

Now it is time to create our RAM disk and create a file system. What we need is a RAM disk, a storage system and terminal services for debugging.

### Add a software service or device stack

1. From the **Software Platform** menu, select **Add...** or click .

   *The Add Software Services and Device Stacks dialog appears. The left side shows the stack services, software services and drivers. The right side shows all peripherals for the selected device. In this tutorial you do not see any peripheral, because we selected a generic architecture.*



Add

Add Software Services and Device Stacks

Select additional software services and stack items

Repository: TASKING POSIX implementation - platform:/plugin/com.tasking.swp.repository.posix/SoftwarePlatform/

Services and Drivers (2/6)

- □ POSIX Message Queues
- □ POSIX Multi-threading
- □ POSIX Signals
- ☑ POSIX Debug Terminal
  - ☑ DBG_TERMINAL_1
- ☑ POSIX Storage System
  - ☑ POSIX_FILESYSTEM_1
- □ POSIX Text Display

Peripherals (0/0)

No peripherals available in selected repository.

Total selection count: 2

**POSIX Storage System**

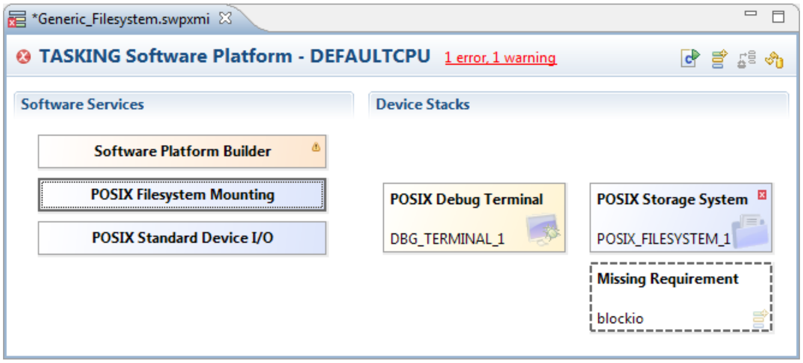
File systems

2. From the **Repository**, select **TASKING POSIX implementation**

   *Only services and drivers from the selected repository are shown.*

3. From the Services and Drivers, select **POSIX Debug Terminal** and **POSIX Storage System** and click **OK**.

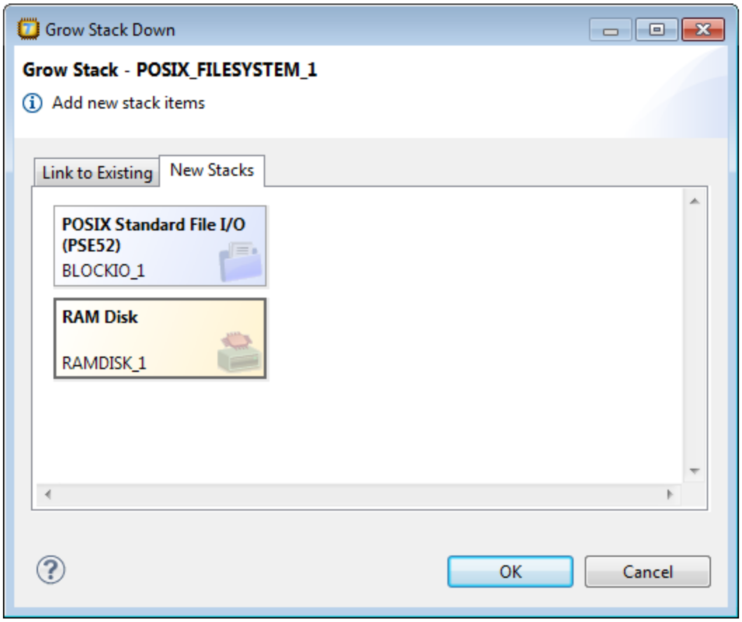
   *The new device stacks and software services appear in the TASKING Software Platform editor. Some software services appear automatically because they are required by other services or stack items.*



As you can see we are missing a requirement. Block device I/O is needed for communication between the RAM Disk and the Storage System. We will add this now.

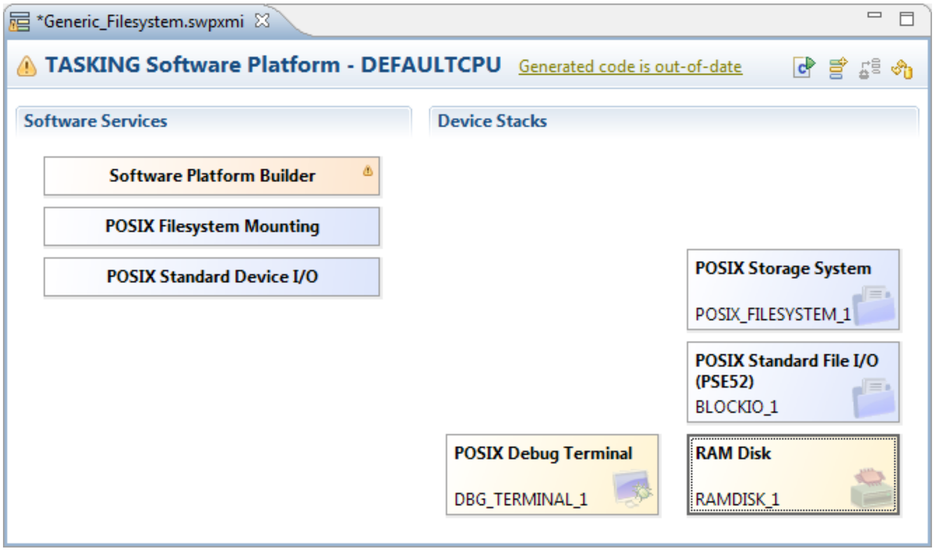
1. Click on the **Missing Requirement** stack item.

   *The Grow Stack dialog appears. The dialog shows all possible combinations of sub-stacks that you can place below the selected module.*



2. We want to add a RAM Disk. To do this, in the **New Stacks** tab, click on **RAM Disk** to select it and click **OK**.

Your Software Platform now looks like this:


*Generic_Filesystem.swpxmi
TASKING Software Platform - DEFAULTCPU
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
POSIX Filesystem Mounting
POSIX Standard Device I/O
POSIX Storage System
POSIX_FILESYSTEM_1
POSIX Standard File I/O (PSE52)
BLOCKIO_1
POSIX Debug Terminal
DBG_TERMINAL_1
RAM Disk
RAMDISK_1

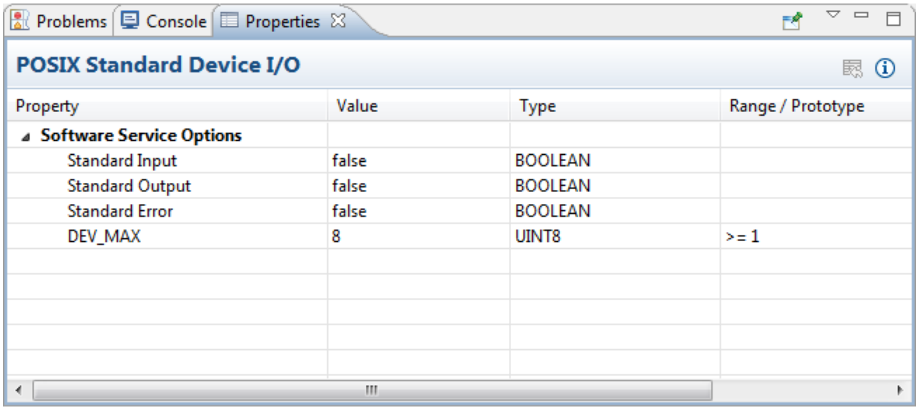
## 4.4. Configure the Software Platform Plug-ins

You can configure peripherals, drivers, stack services and software services. Stack items can have individual options and/or global options. Individual options are only valid for a specific stack item, global options are valid for all stack items of the same type. When you click on an item in the Software Platform Builder, configuration options appear in the Properties view.

In this tutorial we want to redirect Standard Output and Standard Error to the POSIX Debug Terminal.

1. Click on the **POSIX Standard Device I/O** software service.

   *A view similar to the following appears:*

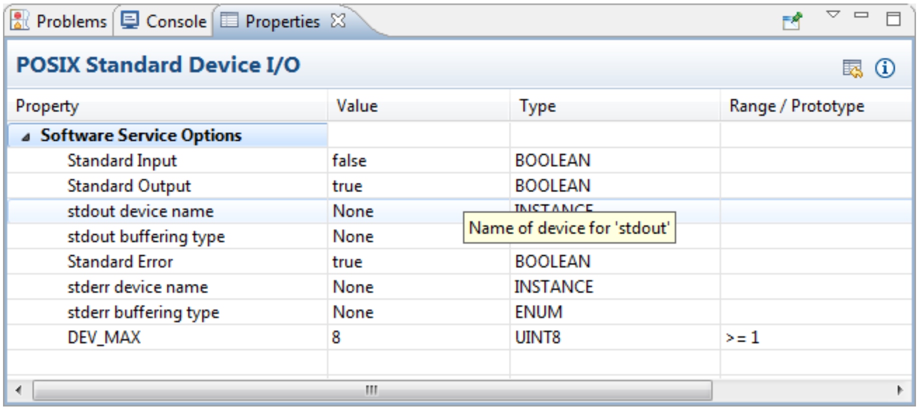


POSIX Standard Device I/O

| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| Software Service Options | | | |
| Standard Input | false | BOOLEAN | |
| Standard Output | false | BOOLEAN | |
| Standard Error | false | BOOLEAN | |
| DEV_MAX | 8 | UINT8 | >= 1 |

2. Change the value of `Standard Output` and `Standard Error` to `true`.

   *New properties for device name and buffer type appear. When you hover the mouse over a property balloon help appears.*

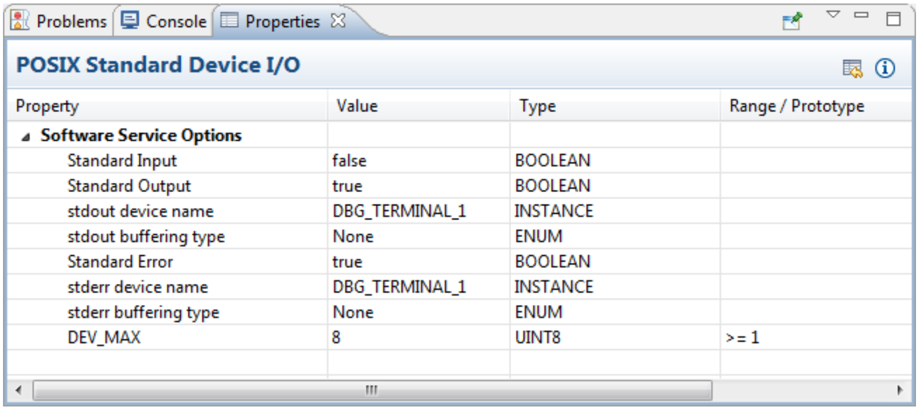


POSIX Standard Device I/O

| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| Software Service Options | | | |
| Standard Input | false | BOOLEAN | |
| Standard Output | true | BOOLEAN | |
| stdout device name | None | INSTANCE | |
| stdout buffering type | None | | |
| Standard Error | true | BOOLEAN | |
| stderr device name | None | INSTANCE | |
| stderr buffering type | None | ENUM | |
| DEV_MAX | 8 | UINT8 | >= 1 |

3. Change the value of `stdout device name` and `stderr device name` to `DBG_TERMINAL_1`. This is the ID of the POSIX Debug Terminal stack item.

   *The properties for POSIX Standard Device I/O should now look like this:*



POSIX Standard Device I/O

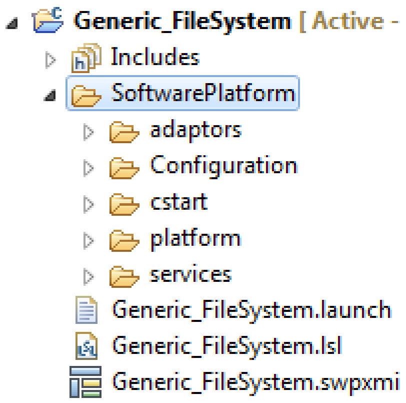
| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| Software Service Options | | | |
| Standard Input | false | BOOLEAN | |
| Standard Output | true | BOOLEAN | |
| stdout device name | DBG_TERMINAL_1 | INSTANCE | |
| stdout buffering type | None | ENUM | |
| Standard Error | true | BOOLEAN | |
| stderr device name | DBG_TERMINAL_1 | INSTANCE | |
| stderr buffering type | None | ENUM | |
| DEV_MAX | 8 | UINT8 | >= 1 |

## 4.5. Generate the Source Code

Now that you have added and configured the Software Platform plug-ins you are ready to generate the Software Platform sources for your project.

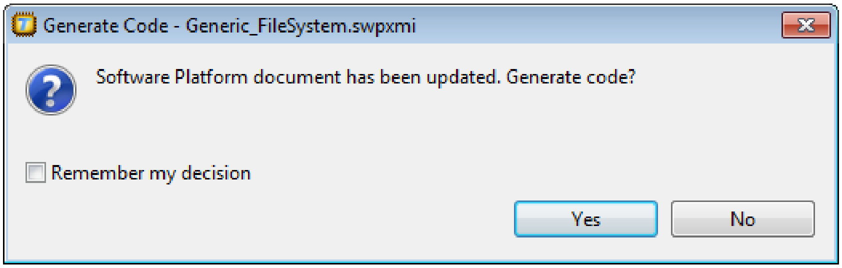
- From the **Software Platform** menu, select **Generate Code** or click .

  *The Software Platform sources are retrieved from the repositories and are added to your project in the folder `SoftwarePlatform`. Tool options such as defines and include paths are added to the Generated* *options field in your project properties. Options can also be set on the* `SoftwarePlatform` *folder and/or sub-folders.*



Generic_FileSystem [ Active - Debug ]
- Includes
- SoftwarePlatform
  - adaptors
  - Configuration
  - cstart
  - platform
  - services
- Generic_FileSystem.launch
- Generic_FileSystem.lsl
- Generic_FileSystem.swpxmi

Also, every time you save the changes you have made to your Software Platform document, a dialog appears asking if you want to generate the code. You can change this behavior in the Preferences dialog as explained in Section 3.7, *Software Platform Builder Preferences*.



Generate Code - Generic_FileSystem.swpxmi

Software Platform document has been updated. Generate code?

□ Remember my decision

## 4.6. Write and Build the Application

Now that the Software Platform sources have been generated we can use it to write an application. In this tutorial we mount a file system on a RAM disk, create a directory, create a file and list the directory contents. This application uses the TASKING POSIX implementation.

1. From the **File** menu, select **New » Source File**.

   *The New Source File dialog appears.*

2. Enter the source file name `main.c` and click **Finish**.

   *The new file* `main.c` *is created and ready for editing in the editor view.*

3. Add the following C source to the file `main.c`.

```
#include <stdio.h>
#include <swplatform.h>
```

```
#include <fs.h>
#include <dirent.h>
#include <sys/stat.h>

#define SRC_DEV "/dev/BLOCKIO_1"
#define DIR_SEP "/"

/* list the directory contents */
static int dirlisting( const char *path )
{
    DIR                *dir;
    struct dirent      *dirent;
    struct stat        buf;
    char               path_to_file[PATH_MAX];

    dir = opendir(path);
    if ( dir != NULL )
    {
        printf( "Contents of folder \"%s\":\n", path );

        while ( 1 )
        {
            dirent = readdir(dir);
            if (dirent == NULL)
            {
                break;
            }
            sprintf(path_to_file, "%s%s%s", path, DIR_SEP, dirent->d_name);

            if (stat(path_to_file, &buf) == 0)
            {
                if (S_ISDIR(buf.st_mode))
                {
                    printf("[DIR]\t%s/\n", dirent->d_name);
                }
                else if (S_ISREG(buf.st_mode))
                {
                    printf("[FILE]\t%s\t(%d bytes)\n", dirent->d_name,
                            (int)(buf.st_size) );
                }
                else
                {
                    printf("[UNK]\t%s\n", dirent->d_name);
                }
            }
        }
        printf( "\n" );

        if (closedir(dir) != 0)
        {
            return -1;
        }
        return 0;
    }
    return -1;
}

int main(void)
{
    int rc;

    /* Create the RAM disk */
    rc = mkfs(SRC_DEV, "fatfs", 0, NULL /* "Ramdisk" */, false);
    if (rc < 0)
    {
        printf("Error: mkfs returned %d\n", rc);
    }

    printf( "Mount filesystem\n" );
    /* Try to mount the first partition ... */
    rc = mount(SRC_DEV, "/ramdisk", "fatfs", 1,
               MOUNT_FLAG_RDWR | MOUNT_FLAG_SYNC);
    if (rc)
    {
        /* ... and if that fails try the entire disk */
        rc = mount(SRC_DEV, "/ramdisk", "fatfs", 0,
                   MOUNT_FLAG_RDWR | MOUNT_FLAG_SYNC);
        if (rc)
        {
            printf("Error: mount returned %d\n", rc);
        }
    }

    dirlisting("/ramdisk");

    printf( "Create  directory\n" );
    rc = mkdir( "/ramdisk/dir1", S_IRWXU | S_IRWXG | S_IRWXO );
    if (rc)
    {
        printf("Error: mkdir returned %d\n", rc);
    }

    FILE *f = fopen("/ramdisk/test.txt", "w");
    if (f == NULL)
    {
        printf("Error: fopen returned NULL\n");
    }
    fwrite("TEST", 4, 1, f);
    fclose(f);
    dirlisting("/ramdisk");
}
```

4. From the **File** menu, select **Save** (Ctrl+S) or click 💾.

   *The file will be saved.*

5. From the **Project** menu, select **Build Generic_Filesystem** or click .

   The project is built and a `Debug` folder is created with the absolute object file `Generic_Filesystem.abs`.

## 4.7. Debug the Application

We are now ready to use the simulator to run and debug the application.

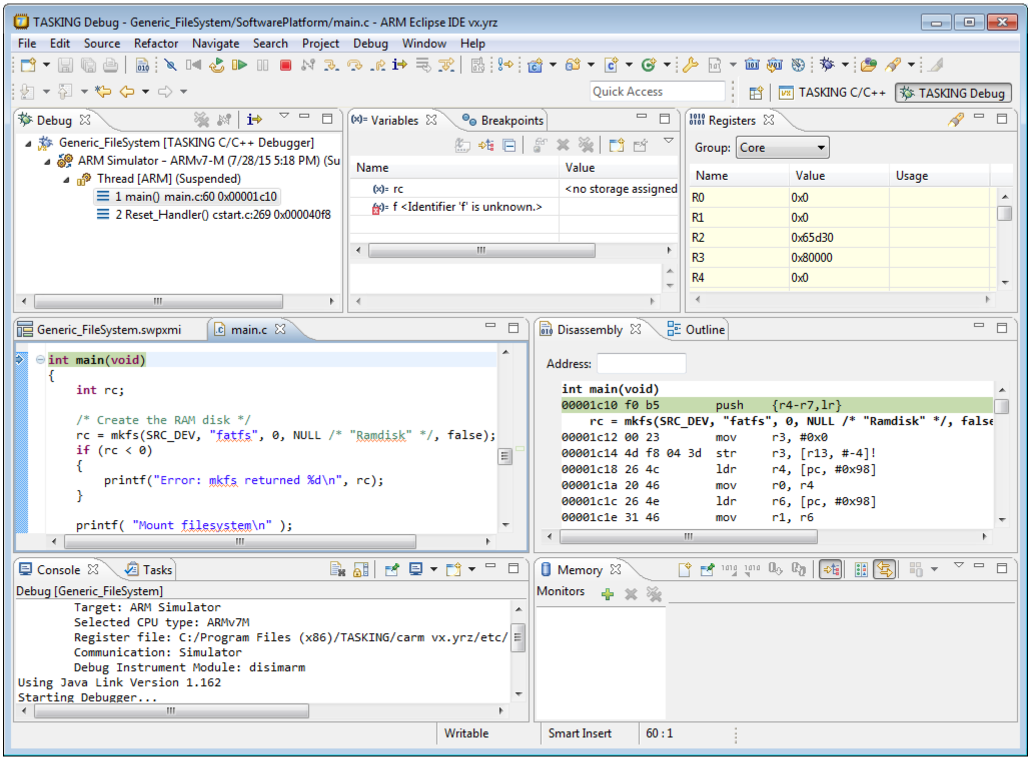
1. From the **Debug** menu select **Debug Generic_Filesystem**.

   Alternatively you can click the 🐞 button in the main toolbar.

   *The TASKING Debug perspective is associated with the TASKING C/C++ Debugger. Because the TASKING C/C++ perspective is still active, Eclipse asks to open the TASKING Debug perspective.*

2. Optionally, enable the option **Remember my decision** and click **Yes**.

   *The debug session is launched. This may take a few seconds. The application suspends when entering the main() function.*


TASKING Debug - Generic_FileSystem/SoftwarePlatform/main.c - ARM Eclipse IDE vx.yrz
File Edit Source Refactor Navigate Search Project Debug Window Help
Quick Access
TASKING C/C++
TASKING Debug
Debug
Variables
Breakpoints
Registers
Generic_FileSystem.swpxmi
main.c
Disassembly
Outline
Console
Tasks
Memory
Monitors
Writable
Smart Insert
60 : 1

```
Generic_FileSystem [TASKING C/C++ Debugger]
  ARM Simulator - ARMv7-M (7/28/15 5:18 PM) (Su
    Thread [ARM] (Suspended)
      1 main() main.c:60 0x00001c10
      2 Reset_Handler() cstart.c:269 0x000040f8
```

| Name | Value |
|---|---|
| rc | <no storage assigned |
| f <Identifier 'f' is unknown.> | |

Group: Core

| Name | Value | Usage |
|---|---|---|
| R0 | 0x0 | |
| R1 | 0x0 | |
| R2 | 0x65d30 | |
| R3 | 0x80000 | |
| R4 | 0x0 | |

```
int main(void)
{
    int rc;

    /* Create the RAM disk */
    rc = mkfs(SRC_DEV, "fatfs", 0, NULL /* "Ramdisk" */, false);
    if (rc < 0)
    {
        printf("Error: mkfs returned %d\n", rc);
    }

    printf( "Mount filesystem\n" );
```

Address:

```
int main(void)
00001c10 f0 b5          push    {r4-r7,lr}
    rc = mkfs(SRC_DEV, "fatfs", 0, NULL /* "Ramdisk" */, false
00001c12 00 23          mov     r3, #0x0
00001c14 4d f8 04 3d    str     r3, [r13, #-4]!
00001c18 26 4c          ldr     r4, [pc, #0x98]
00001c1a 20 46          mov     r0, r4
00001c1c 26 4e          ldr     r6, [pc, #0x98]
00001c1e 31 46          mov     r1, r6
```

```
Debug [Generic_FileSystem]
          Target: ARM Simulator
          Selected CPU type: ARMv7M
          Register file: C:/Program Files (x86)/TASKING/carm vx.yrz/etc/
          Communication: Simulator
          Debug Instrument Module: disimarm
Using Java Link Version 1.162
Starting Debugger...
```

3. To resume execution, from the **Debug** menu, select **Resume**, or press **F8**, or click on the **Resume** button (▶).

   *The application is executed. The FSS view now shows:*

```
Mount filesystem
Contents of folder "/ramdisk":

Create  directory
Contents of folder "/ramdisk":
[DIR]   DIR1/
[FILE]  TEST.TXT        (4 bytes)
```

Congratulations, you just made an example Software Platform application.

For more information about debugging an application see Chapter *Debugging your Application* of the *Getting Started with the TASKING VX-toolset*, and Chapter *Using the Debugger* of the *TASKING VX-toolset for ARM User Guide*.

# Chapter 5. Tutorial Using a Library Wrapper

In this tutorial we show how to use the Software Platform as a wrapper to access library functions from STMicroelectronics. You will learn how to use a UART for communication over a serial port. This example uses the STM3210E-EVAL evaluation board.

## 5.1. Create a Project

As explained in the previous chapter, the first step is to create a C project.

1. From the **File** menu, select **New » TASKING ARM C/C++ Project**

   *The New C/C++ Project wizard appears.*

2. Enter a name for your project, for example `USART_printf`.

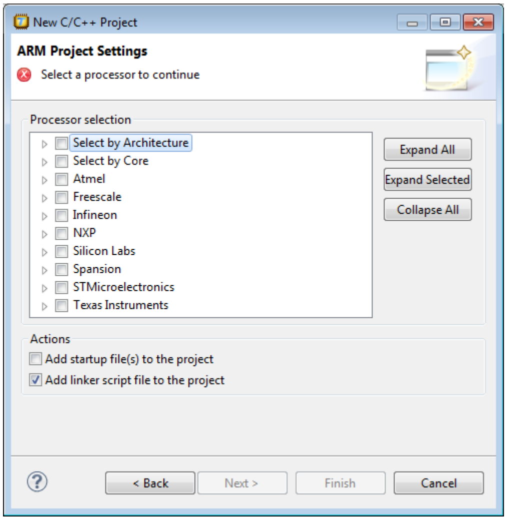
   *In the **Location** field you will see the location where the new project will be stored.*

3. In the **Project type** box you can select whether to create an application or a library.

   - Expand **TASKING ARM Application** and select **Empty Project**. This creates a project without a C source file.

   - Click **Next** to continue.

   *The ARM Project Settings page appears.*



New C/C++ Project

ARM Project Settings

Select a processor to continue

Processor selection

- ☐ Select by Architecture
- ☐ Select by Core
- ☐ Atmel
- ☐ Freescale
- ☐ Infineon
- ☐ NXP
- ☐ Silicon Labs
- ☐ Spansion
- ☐ STMicroelectronics
- ☐ Texas Instruments

Actions

☐ Add startup file(s) to the project

☑ Add linker script file to the project

4. Select the target processor core or architecture for which you want to build the application. In this example, we select **STM32F103ZG**. Afterwards you can always change the processor in the **Project » Properties for** dialog.

5. Leave the rest of the dialog as is and click **Next**.

   *The Target Settings page appears.*

6. In order to debug your project you need to create a debug configuration.

   - Select a target. For this tutorial select **STMicroelectronics STM3210E-Eval board**.
   - For connection select **J-Link over USB** or **ST-LINK over USB**.
   - Enable **Add launch configuration to the project**. This allows you to debug your project.
   - Leave the other tabs as is. For more information, see section *Creating a Customized Debug Configuration* in Chapter *Using the Debugger* of the *TASKING VX-toolset for ARM User Guide*.

7. Click **Finish** to finish the wizard and to create the project.

   *The project has now been created and is the active project.*

## 5.2. Add a Software Platform Document

Now that we have a project we can add a Software Platform document.

1. From the **File** menu, select **New » TASKING Software Platform Document**.

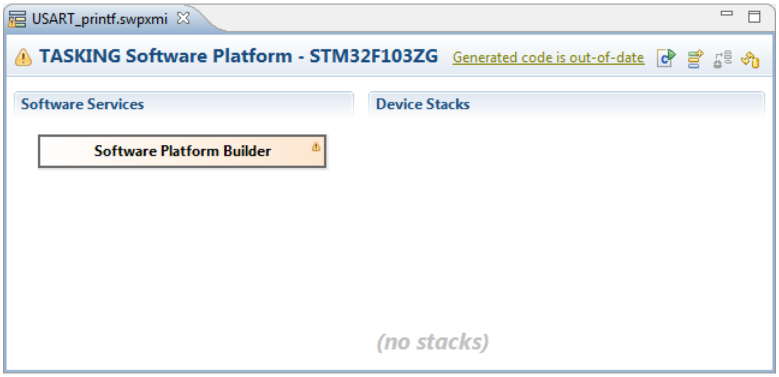
   *The New Software Platform Document dialog appears.*

2. Leave the project name and file name as is and click **Finish**.

   *A new Software Platform document is added to the active project (`USART_printf.swpxmi`).*

### TASKING Software Platform Builder editor

Double-click on the Software Platform document `USART_printf.swpxmi` to open it. The editor consists of two sections: the **Software Services** and the **Device Stacks**.


USART_printf.swpxmi
TASKING Software Platform - STM32F103ZG
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
(no stacks)

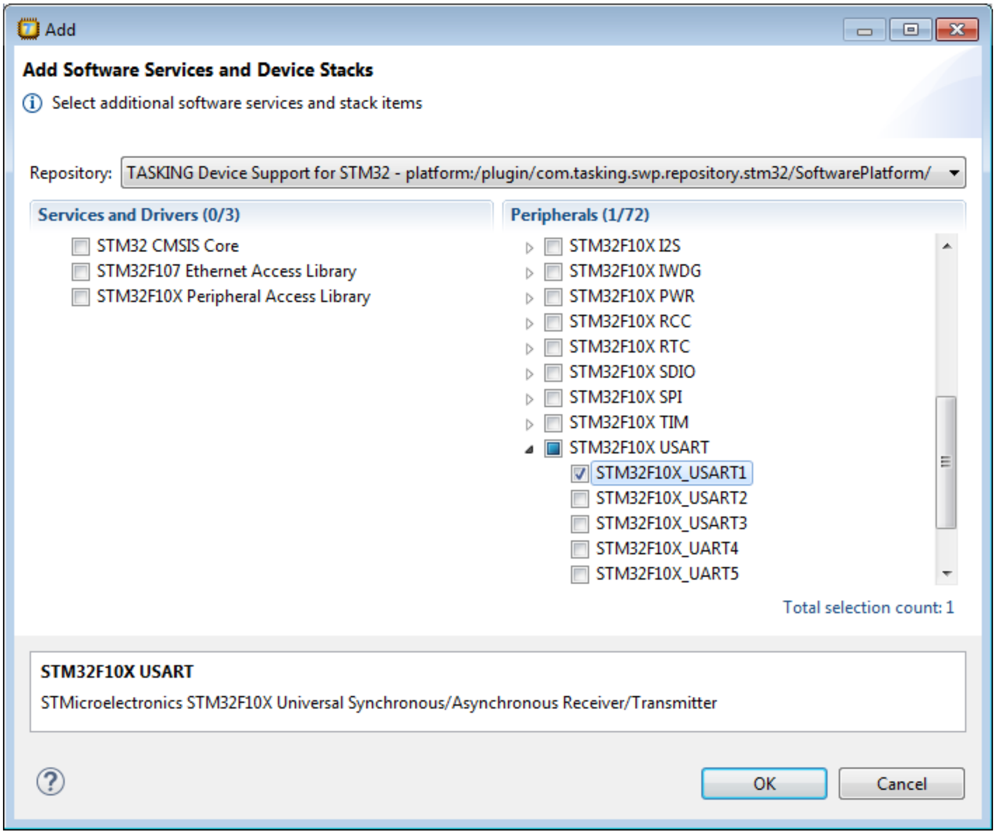
# 5.3. Add the Services and Device Stacks

Now it is time to add a USART peripheral and connect a library wrapper to it.

## Add a software service or device stack

1. From the **Software Platform** menu, select **Add...** or click .

   *The Add Software Services and Device Stacks dialog appears. The left side shows the stack services, software services and drivers. The right side shows all peripherals for the selected device.*



**Add Software Services and Device Stacks**

Select additional software services and stack items

Repository: TASKING Device Support for STM32 - platform:/plugin/com.tasking.swp.repository.stm32/SoftwarePlatform/

Services and Drivers (0/3)

- □ STM32 CMSIS Core
- □ STM32F107 Ethernet Access Library
- □ STM32F10X Peripheral Access Library

Peripherals (1/72)

- □ STM32F10X I2S
- □ STM32F10X IWDG
- □ STM32F10X PWR
- □ STM32F10X RCC
- □ STM32F10X RTC
- □ STM32F10X SDIO
- □ STM32F10X SPI
- □ STM32F10X TIM
- ■ STM32F10X USART
  - ☑ STM32F10X_USART1
  - □ STM32F10X_USART2
  - □ STM32F10X_USART3
  - □ STM32F10X_UART4
  - □ STM32F10X_UART5

Total selection count: 1

**STM32F10X USART**

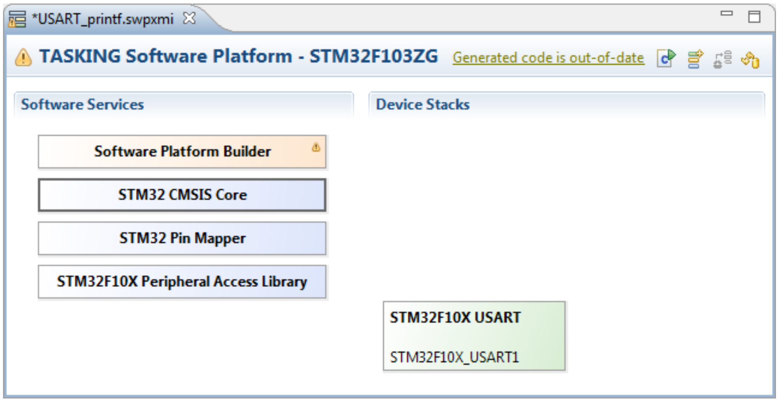
STMicroelectronics STM32F10X Universal Synchronous/Asynchronous Receiver/Transmitter

2. From the **Repository**, select **TASKING Device Support for STM32**

   *Only services and drivers and peripherals from the selected repository are shown.*

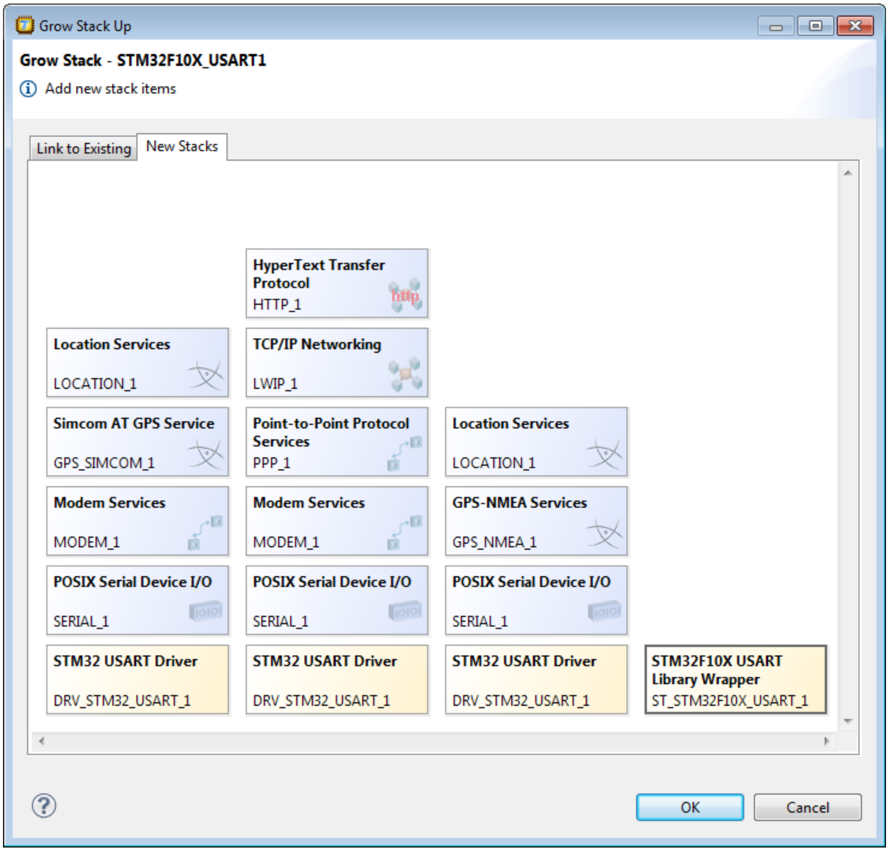
3. From the Peripherals, select **STM32F10X_USART1** and click **OK**.

   *The new device stack and software service appear in the TASKING Software Platform editor. The STM32 CMSIS Core, STM32 Pin Mapper and STM32F10X Peripheral Access Library software services appear automatically because it is required by the peripheral.*


*USART_printf.swpxmi
TASKING Software Platform - STM32F103ZG
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
STM32 CMSIS Core
STM32 Pin Mapper
STM32F10X Peripheral Access Library
STM32F10X USART
STM32F10X_USART1

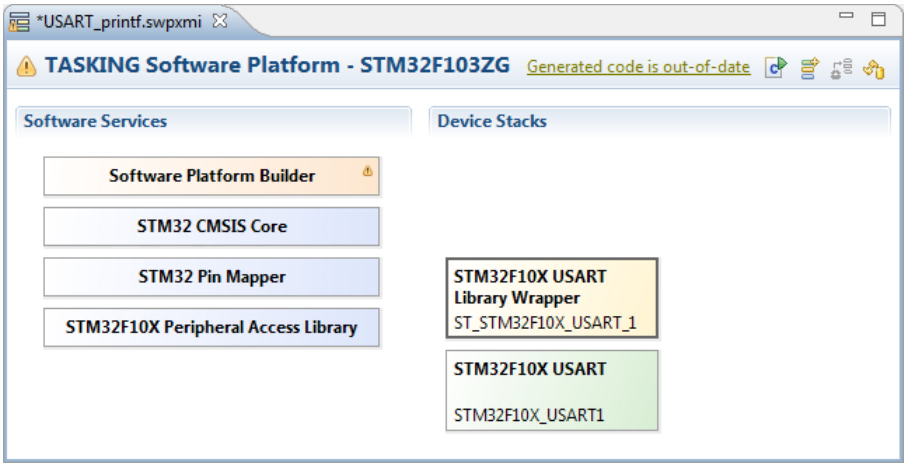
4. Right-click on the **STM32F10X USART** stack item and select **Grow Stack Up** from the pop-up menu.

   *The Grow Stack dialog appears. The dialog shows all possible combinations of sub-stacks that you can place on top of the selected module.*



5. We want to access the STMicroelectronics functions that are present in the Software Platform repository by means of a library wrapper. To do this, in the **New Stacks** tab, click on **STM32F10X USART Library Wrapper** to select it and click **OK**.

Your Software Platform now looks like this:



*USART_printf.swpxmi

TASKING Software Platform - STM32F103ZG Generated code is out-of-date

Software Services

Software Platform Builder

STM32 CMSIS Core

STM32 Pin Mapper

STM32F10X Peripheral Access Library

Device Stacks

STM32F10X USART Library Wrapper
ST_STM32F10X_USART_1

STM32F10X USART
STM32F10X_USART1

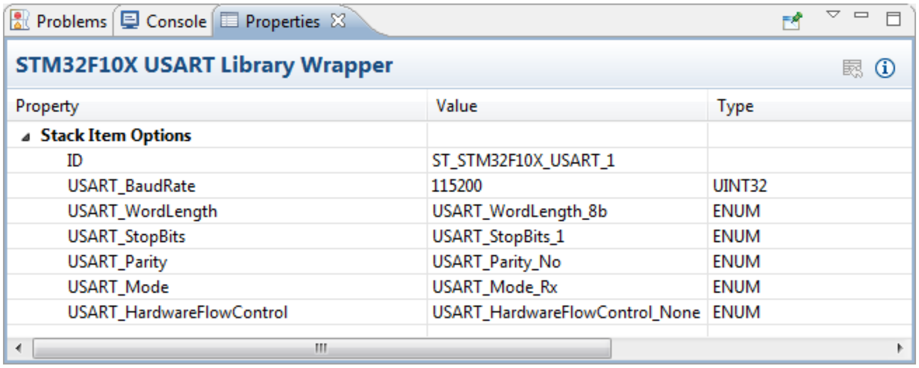
## 5.4. Configure the Software Platform Plug-ins

You can configure peripherals, drivers, stack services and software services. Stack items can have individual options and/or global options. Individual options are only valid for a specific stack item, global options are valid for all stack items of the same type. When you click on an item in the Software Platform Builder, configuration options appear in the Properties view.

In this tutorial we want to change the USART communication mode of the library wrapper.

1. Click on the **STM32F10X USART Library Wrapper** stack item.

   *A view similar to the following appears:*



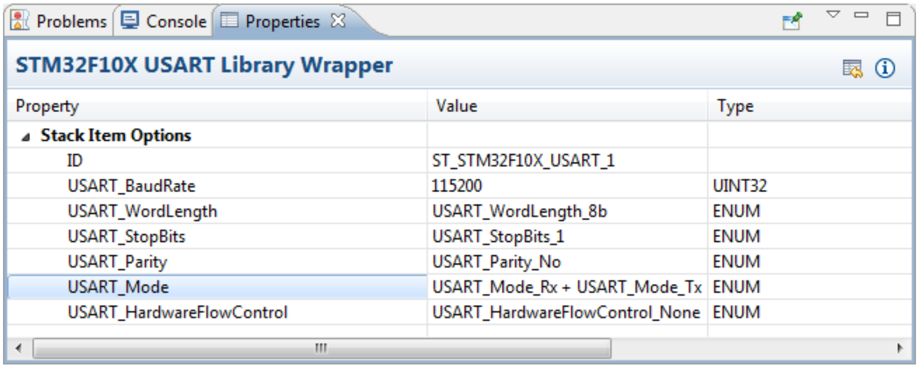
Problems | Console | Properties

STM32F10X USART Library Wrapper

| Property | Value | Type |
|---|---|---|
| Stack Item Options | | |
| ID | ST_STM32F10X_USART_1 | |
| USART_BaudRate | 115200 | UINT32 |
| USART_WordLength | USART_WordLength_8b | ENUM |
| USART_StopBits | USART_StopBits_1 | ENUM |
| USART_Parity | USART_Parity_No | ENUM |
| USART_Mode | USART_Mode_Rx | ENUM |
| USART_HardwareFlowControl | USART_HardwareFlowControl_None | ENUM |

2. Change the value of `USART_Mode` to `USART_Mode_Rx + USART_Mode_Tx`.

   *The properties for STM32F10X USART Library Wrapper should now look like this:*



**STM32F10X USART Library Wrapper**

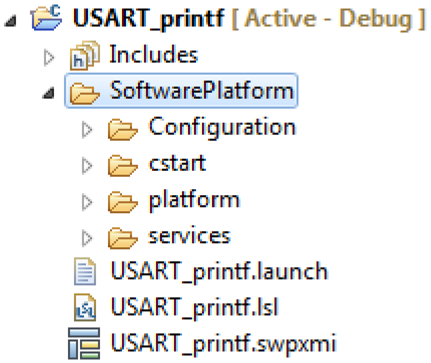
| Property | Value | Type |
|---|---|---|
| **Stack Item Options** | | |
| ID | ST_STM32F10X_USART_1 | |
| USART_BaudRate | 115200 | UINT32 |
| USART_WordLength | USART_WordLength_8b | ENUM |
| USART_StopBits | USART_StopBits_1 | ENUM |
| USART_Parity | USART_Parity_No | ENUM |
| USART_Mode | USART_Mode_Rx + USART_Mode_Tx | ENUM |
| USART_HardwareFlowControl | USART_HardwareFlowControl_None | ENUM |

## 5.5. Generate the Source Code

Now that you have added and configured the Software Platform plug-ins you are ready to generate the Software Platform sources for your project.

1. From the **Software Platform** menu, select **Generate Code** or click .

   *The Software Platform sources are retrieved from the repositories and are added to your project in the folder `SoftwarePlatform`. Tool options such as defines and include paths are added to the Generated options field in your project properties. Options can also be set on the `SoftwarePlatform` folder and/or sub-folders.*



```
USART_printf [ Active - Debug ]
    Includes
    SoftwarePlatform
        Configuration
        cstart
        platform
        services
    USART_printf.launch
    USART_printf.lsl
    USART_printf.swpxmi
```

2. From the **File** menu, select **Save** (Ctrl+S) or click .

   *The Software Platform document will be saved.*

## 5.6. Configure the Pins

Now that the Software Platform software services and device stacks are defined, you need to configure the pins of the peripheral. In this case the USART. You can do this with the TASKING Pin Mapper.

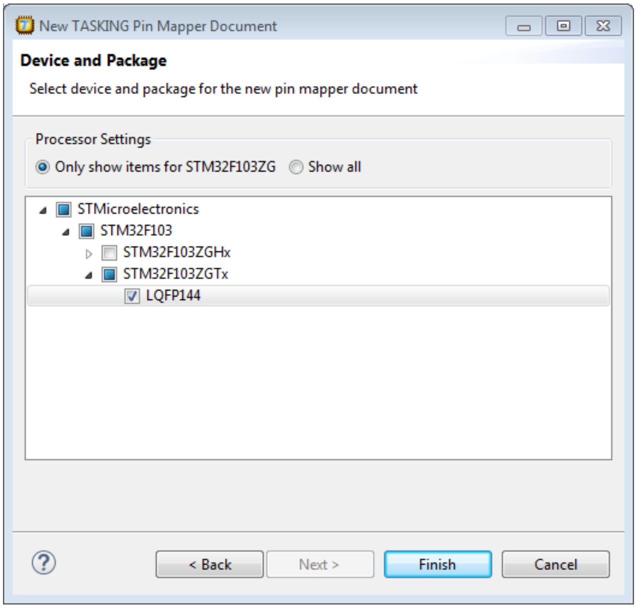
1. From the **File** menu, select **New » TASKING Pin Mapper Document**.

   *The New TASKING Pin Mapper Document wizard appears.*

2. Select the **Project** folder for the Pin Mapper document: type the name of your project (`USART_printf`) or click the **Browse** button to select a project.

3. In the **File name** field, enter a name for the Pin Mapper document, for example `USART_printf.pincfg` and click **Next**.

   *The Device and Package page appears.*



New TASKING Pin Mapper Document

**Device and Package**

Select device and package for the new pin mapper document

Processor Settings

● Only show items for STM32F103ZG ○ Show all

- ▪ STMicroelectronics
  - ▪ STM32F103
    - □ STM32F103ZGHx
    - ▪ STM32F103ZGTx
      - ☑ LQFP144

4. Select the processor and package that is used on the board and click **Finish**. In our example, we use the STM3210E-Eval board, which has a STM32F103ZGTx and package LQFP144.

   *A pin configuration file with extension `.pincfg` is added to the existing project. Because the TASKING C/C++ perspective is still active, Eclipse asks to open the TASKING Pin Mapper perspective.*

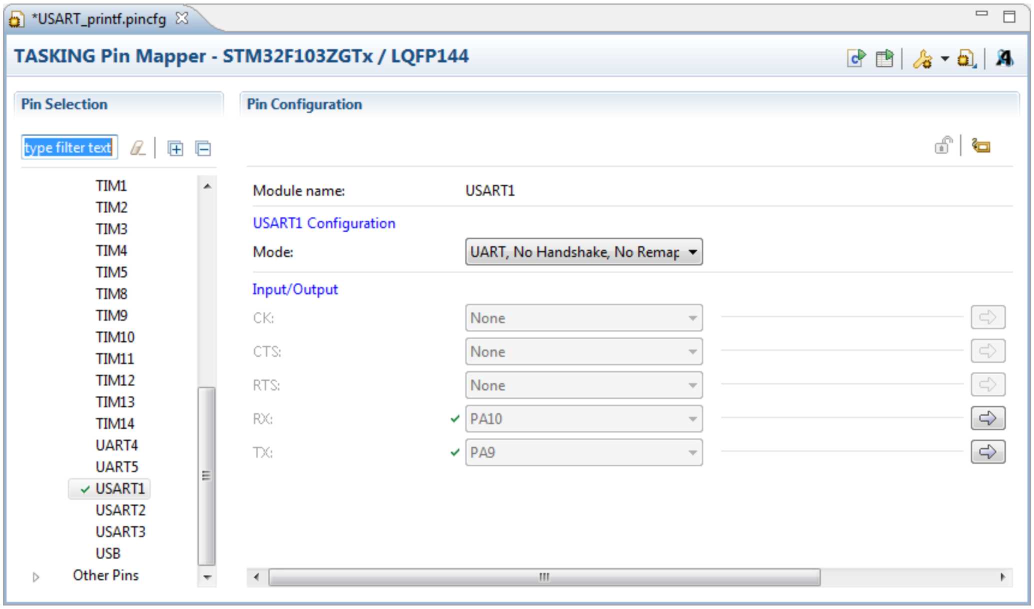
5. Optionally, enable the option **Remember my decision** and click **Yes**.

   *The TASKING Pin Mapper perspective is opened.*

6. In the Pin Selection pane, select **Peripherals » USART1**.

7. In the Pin Configuration pane, set the **Mode** to **UART, No Handshake, No Remap**.

   *This selection automatically connects RX to PA10 and TX to PA9. A green check mark ✓ appears if the connection is successful.*



Module name: USART1

USART1 Configuration

Mode: UART, No Handshake, No Remap

Input/Output

CK: None

CTS: None

RTS: None

RX: ✓ PA10

TX: ✓ PA9

8. From the **File** menu, select **Save** (Ctrl+S) or click  to save the configuration.

9. From the **Pin Mapper** menu, select **Generate Code** or click .

   *The Pin Mapper sources are generated and are added to your project in the folder `PinMapper`.*

10. Return to the TASKING C/C++ perspective.

## 5.7. Write and Build the Application

Now that the Software Platform sources have been generated we can use it to write an application. In this tutorial we use the USART port on the evaluation board to send a message to the PC. This application uses the library wrapper function `stm32_usart_open` to configure the USART as specified by the stack item options, which makes it easier to access the USART.

1. From the **File** menu, select **New » Source File**.

   *The New Source File dialog appears.*

2. Enter the source file name `main.c` and click **Finish**.

   *The new file `main.c` is created and ready for editing in the editor view.*

3. Add the following C source to the file `main.c`.

   ```
   #include <stdio.h>
   #include "devices.h"
   #include "st_stm32_usart.h"

   USART_TypeDef * usart = NULL;

   int main(void)
   {
       /* initialize the USART device using the library wrapper  */
       usart = stm32_usart_init(ST_STM32F10X_USART_1);

       printf("Hello world !!!\n");

       return 0;
   }

   /* use STM32F0XX peripheral access library functions */
   int fputc(int ch, FILE *f)
   {
       /* check transmit data register empty flag */
       while (USART_GetFlagStatus(usart, USART_FLAG_TXE) != SET);

       /* send one character */
       USART_SendData(usart, (uint8_t)ch);

       return ch;
   }
   ```

4. From the **File** menu, select **Save** (Ctrl+S) or click 💾.

   *The file will be saved.*

5. From the **Project** menu, select **Build USART_printf** or click 101.

   The project is built and a `Debug` folder is created with the absolute object file `USART_printf.abs`.

## 5.8. Connect the Board

Before you can debug the program you need to connect the STM3210E-EVAL board. The STM3210E-EVAL is an evaluation board for STMicroelectronics ARM Cortex-M3 core-based STM32F103Z microcontrollers. Read the *STM3210E-EVAL evaluation board User Manual (UM0488)* for detailed information about the board.

## Installing driver software

Before you connect J-Link or ST-LINK to the board, first make sure you have installed the latest driver software. You can download the latest software for J-Link from www.segger.com/download_jlink.html. You can download the latest software for ST-LINK from:

http://www.st.com/web/en/catalog/tools/PF258167 (Windows 7, XP),

http://www.st.com/web/en/catalog/tools/PF259459 (Windows 8).

## Jumper settings

Set jumper JP13 to PSU for power through the 5V DC power jack.

## Connecting the board

1. Connect any of the J-Link or ST-LINK debugger probes with the 20-pin JTAG/SWD cable to the board.

2. Connect J-Link or ST-LINK to a PC with a USB cable.

3. Connect port USART1 with a serial cable to a PC.

4. Power the board with the 5V DC power adapter connected to the power jack on the board (CN17).

# 5.9. Debug the Application

We are now ready to use the evaluation board to run and debug the application.

To see the output of the application on your PC you can use a terminal emulator.

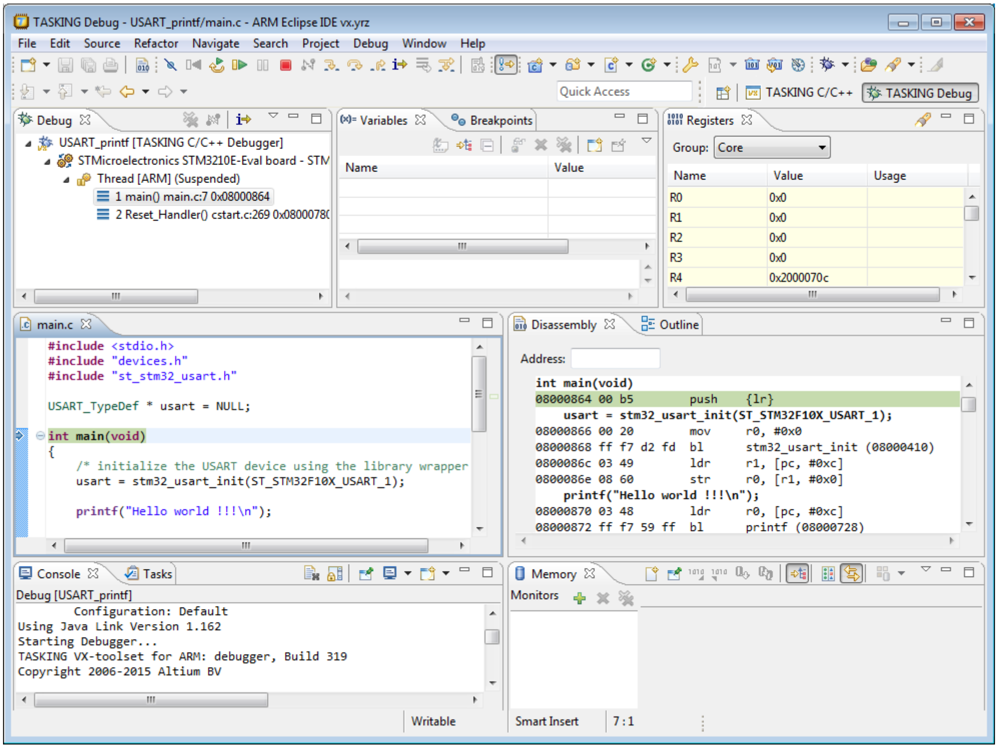
1. From the **Debug** menu select **Debug USART_printf**.

   Alternatively you can click the 🐞 button in the main toolbar.

   *The TASKING Debug perspective is associated with the TASKING C/C++ Debugger. Because the TASKING C/C++ perspective is still active, Eclipse asks to open the TASKING Debug perspective.*

2. Optionally, enable the option **Remember my decision** and click **Yes**.

   *The debug session is launched. This may take a few seconds. The application suspends when entering the main() function.*



```
USART_printf [TASKING C/C++ Debugger]
  STMicroelectronics STM3210E-Eval board - STM
    Thread [ARM] (Suspended)
      1 main() main.c:7 0x08000864
      2 Reset_Handler() cstart.c:269 0x0800078(
```

Variables:

| Name | Value |
|---|---|

Registers — Group: Core

| Name | Value | Usage |
|---|---|---|
| R0 | 0x0 | |
| R1 | 0x0 | |
| R2 | 0x0 | |
| R3 | 0x0 | |
| R4 | 0x2000070c | |

main.c:

```
#include <stdio.h>
#include "devices.h"
#include "st_stm32_usart.h"

USART_TypeDef * usart = NULL;

int main(void)
{
    /* initialize the USART device using the library wrapper
    usart = stm32_usart_init(ST_STM32F10X_USART_1);

    printf("Hello world !!!\n");
```

Disassembly — Address:

```
int main(void)
08000864 00 b5          push    {lr}
    usart = stm32_usart_init(ST_STM32F10X_USART_1);
08000866 00 20          mov     r0, #0x0
08000868 ff f7 d2 fd    bl      stm32_usart_init (08000410)
0800086c 03 49          ldr     r1, [pc, #0xc]
0800086e 08 60          str     r0, [r1, #0x0]
    printf("Hello world !!!\n");
08000870 03 48          ldr     r0, [pc, #0xc]
08000872 ff f7 59 ff    bl      printf (08000728)
```

Console — Debug [USART_printf]

```
          Configuration: Default
Using Java Link Version 1.162
Starting Debugger...
TASKING VX-toolset for ARM: debugger, Build 319
Copyright 2006-2015 Altium BV
```

3. To resume execution, from the **Debug** menu, select **Resume**, or press **F8**, or click on the **Resume** button (▶).

   *The application is executed. When you have a terminal emulator open on your PC, this shows:*

   ```
   Hello world !!!
   ```

For more information about debugging an application see Chapter *Debugging your Application* of the *Getting Started with the TASKING VX-toolset*, and Chapter *Using the Debugger* of the *TASKING VX-toolset for ARM User Guide*.

# Chapter 6. Tutorial Creating a Local Repository

In this tutorial we show how to create your own Software Platform repository. This can be useful when you want to create your own plug-ins or to adapt existing plug-ins to suit your needs.

We assume that the `Generic_Filesystem` project of Chapter 4, *Tutorial Creating a File System on a RAM Disk* is still present in the C/C++ Projects view.

## 6.1. Create a Local Repository Project

To use a local repository we need to create a project. This does not have to be a toolset project, because we do not have to compile anything. It is just a container for your local plug-ins. Therefore, it can be a general project.

1. From the **File** menu, select **New » Project**

   *The New Project wizard appears.*

2. Select **General » Project** and click **Next**.

   *The Project page appears.*

3. Enter a name for your project, for example `My_Repository`.

   *In the **Location** field you will see the location where the new project will be stored.*

4. Click **Finish** to finish the wizard and to create the project.

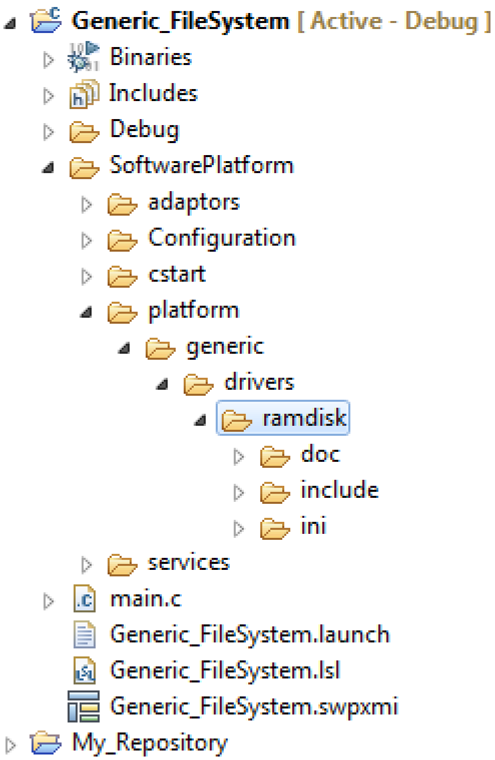
   *The project has now been created and is visible in the C/C++ Projects view.*

## 6.2. Add a Software Platform Plug-in

You can copy existing plug-ins as a starting point, or you can write one from scratch. In this tutorial we make a copy of a plug-in that was generated in the tutorial of Chapter 4, *Tutorial Creating a File System on a RAM Disk*.

1. From the `Generic_Filesystem` project, right-click on the `ramdisk` folder in `SoftwarePlatform/platform/generic/drivers` and select **Copy**.

   > The exact path depends on the directory structure of the Software Platform repository from which the plug-in is retrieved. You can find out the correct path when you click the **Show Resource Info** button (ⓘ) in the Properties view of the RAM Disk.



```
Generic_FileSystem [ Active - Debug ]
    Binaries
    Includes
    Debug
    SoftwarePlatform
        adaptors
        Configuration
        cstart
        platform
            generic
                drivers
                    ramdisk
                        doc
                        include
                        ini
        services
    main.c
    Generic_FileSystem.launch
    Generic_FileSystem.lsl
    Generic_FileSystem.swpxmi
My_Repository
```

2. Right-click on the `My_Repository` project and select **New » Folder**.

   *The New Folder dialog appears.*

3. Select the `My_Repository` parent folder if it has not been selected already.

4. In the **Folder name** field enter a name to categorize the type of plug-ins. For this tutorial, enter `My_customized_plugins` and click **Finish**.

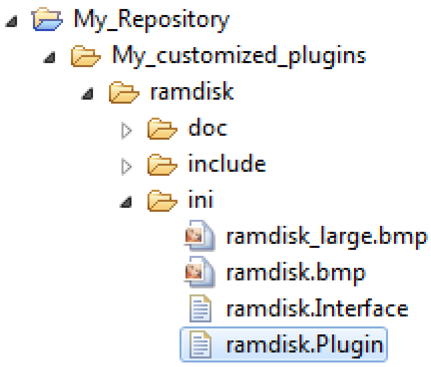
5. Right-click on the `My_customized_plugins` folder and select **Paste**.

   *A copy of the ramdisk plug-in is now present in the local repository.*

## 6.3. Modify the Software Platform Plug-in

As long as the name of a plug-in does not change, the new plug-in overrules the plug-in in the default repository. In this tutorial we will only alter the caption (the name of the stack item) of the RAM disk plug-in we just copied.

1. Open the `My_Repository` project and open the `ramdisk.Plugin` file.



- My_Repository
  - My_customized_plugins
    - ramdisk
      - doc
      - include
      - ini
        - ramdisk_large.bmp
        - ramdisk.bmp
        - ramdisk.Interface
        - ramdisk.Plugin

*The file `ramdisk.Plugin` appears in the editor view.*

2. Under the `[Plugin]` part, change the line

   ```
   Caption=RAM Disk
   ```

   into

   ```
   Caption=My RAM Disk
   ```

   If a question appears that the file is read-only and if you wish to make it writable, click **Yes**.

3. From the **File** menu, select **Save** (Ctrl+S) or click 💾.

   *The file will be saved.*

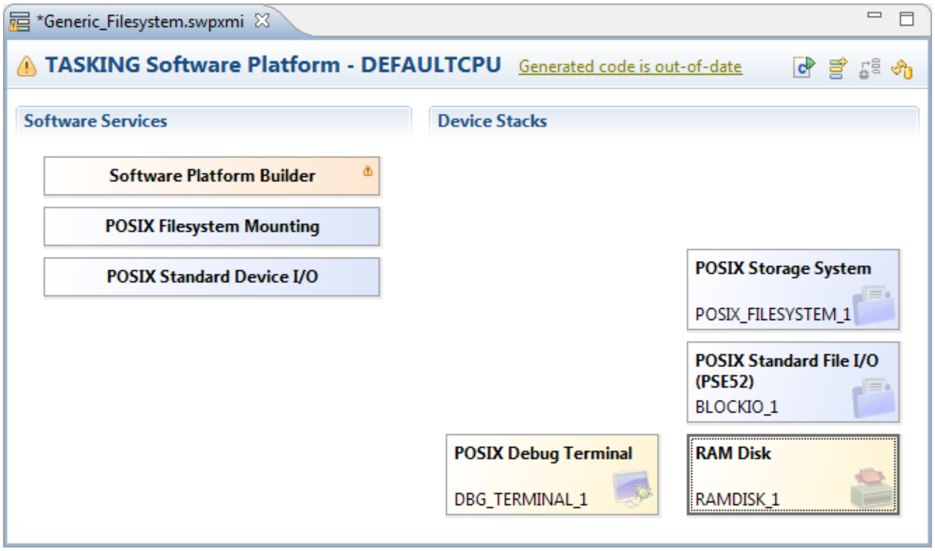
## 6.4. Use the Local Repository

Now that we have a local repository, we want to use it in the `Generic_Filesystem` project that we created in Chapter 4, *Tutorial Creating a File System on a RAM Disk*.

You can use Eclipse platform URLs or a file URL to specify the location of a repository or you can use a browse button. The following URLs are supported.

| URL | Description |
|---|---|
| `platform:/resource` | This URL is used to identify a resource located in the workspace. The next path segment after "resource" should be the name of a project, which can be followed by the folder and/or file we want to locate. |
| `platform:/plugin` | This URL is used to locate a resource available in a Eclipse plug-in/bundle. The path segment after "plugin" should be the identifier of the bundle, which can be followed by the path of the resource in the bundle. This URL is used internally for the default Software Platform repository. |
| `file://` | This URL is used to specify a local file or directory. |

1. In the `Generic_Filesystem` project, open the Software Platform document `Generic_Filesystem.swpxmi`.

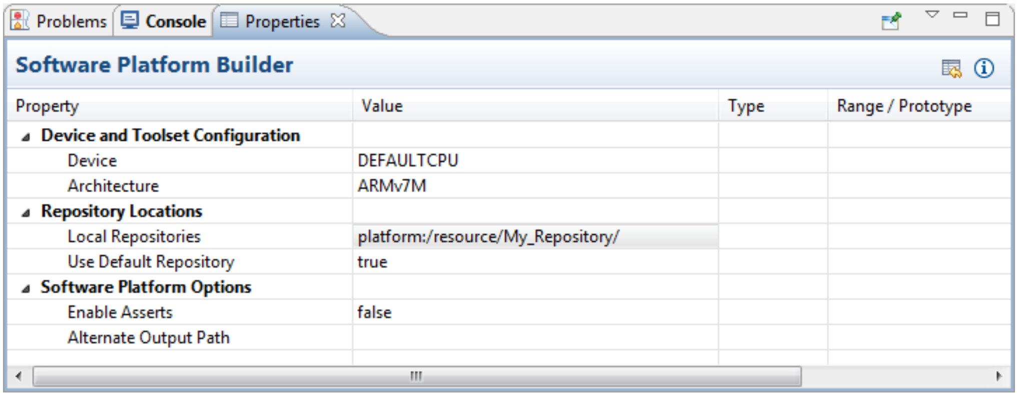

*Generic_Filesystem.swpxmi
TASKING Software Platform - DEFAULTCPU
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
POSIX Filesystem Mounting
POSIX Standard Device I/O
POSIX Storage System
POSIX_FILESYSTEM_1
POSIX Standard File I/O (PSE52)
BLOCKIO_1
POSIX Debug Terminal
DBG_TERMINAL_1
RAM Disk
RAMDISK_1

2. Select the **Software Platform Builder** software service.

   *The Software Platform Builder Properties view appears.*

3. Change the value of `Local Repositories` to `platform:/resource/My_Repository`. Note that this field is case sensitive.

   You can enter the URL directly or you can click the ... button to open the Local Repository Locations. Click **Add** to open the Select Location dialog. Click **Workspace** to open the Folder Selection dialog. Select `My_Repository` and click **OK** in all dialogs.

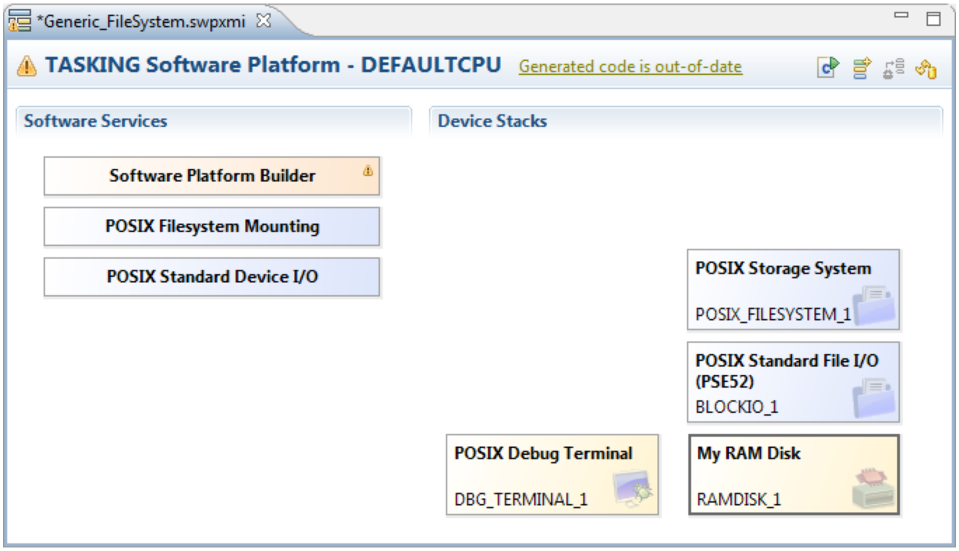


| Property | Value | Type | Range / Prototype |
|---|---|---|---|
| Device and Toolset Configuration | | | |
| Device | DEFAULTCPU | | |
| Architecture | ARMv7M | | |
| Repository Locations | | | |
| Local Repositories | platform:/resource/My_Repository/ | | |
| Use Default Repository | true | | |
| Software Platform Options | | | |
| Enable Asserts | false | | |
| Alternate Output Path | | | |

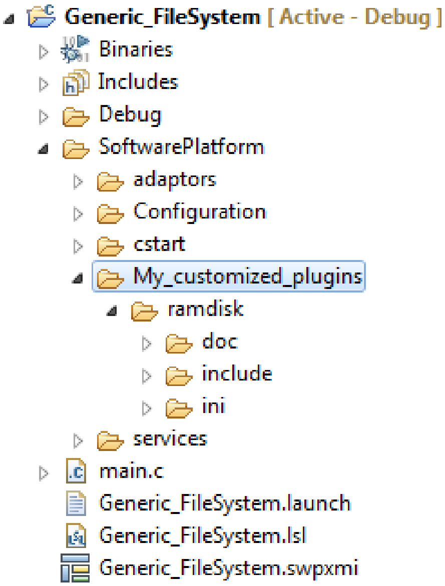
*An error appears that the Repository cache is out-of-date.*

4. From the **Software Platform** menu, select **Reload Repository** or click .

   *As you can see, the RAM Disk stack item has been replaced by My RAM Disk.*


*Generic_FileSystem.swpxmi
TASKING Software Platform - DEFAULTCPU
Generated code is out-of-date
Software Services
Device Stacks
Software Platform Builder
POSIX Filesystem Mounting
POSIX Standard Device I/O
POSIX Storage System
POSIX_FILESYSTEM_1
POSIX Standard File I/O (PSE52)
BLOCKIO_1
POSIX Debug Terminal
DBG_TERMINAL_1
My RAM Disk
RAMDISK_1

If you would generate code again (), the existing plug-in in the generated `SoftwarePlatform` folder is replaced by the plug-in from `My_Repository`. As you can see the `My_customized_plugins` folder has been added to the `Generic_FileSystem` project.



# Chapter 7. Content of the Default ARM Repositories

The chapter describes the content of the default ARM repositories.

When Altium releases a new version of a repository you can update this repository by using the **Available Updates** wizard.

### To check for updates

1. From the **Help** menu, select **Check for Updates**.

   *The Available Updates wizard appears.*

2. Follow the steps in the wizard and click **Finish**.

### Online repository documentation

Each repository contains detailed descriptions of its contents. You can access this documentation from the Help menu (**Help » Help Contents » Software Platform Repository Reference - ...**).

## 7.1. TASKING POSIX Repository

The Portable Operating System Interface POSIX® is a set of well-established standards that define a standardized way for an application to interface to the operating system. Because of the aim of portability and readability we have chosen POSIX as our standard interfaces at application level for at least the kernel (multithreading), device I/O and networking services.

The POSIX standards do not specify how services must be implemented, just their semantics. Implementers can choose their implementation as long as they follow the specification of the interface. These standards have become the de-facto programming standards also for embedded environments.

The complete set of POSIX services as defined in the standard may be useful for large applications, but the set is considered to be too large for most embedded systems. Embedded systems usually have tight memory requirements, may not have memory management capabilities, and may not even have a secondary memory for implementing the UNIX file system.

For these reasons the POSIX standard recognizes the need for the creation of subsets of the operating system services. The IEEE Std 1003.13-2003 edition (POSIX.13) describes four real-time application environment profiles and their minimum hardware requirements.

The TASKING POSIX implementation conforms to the PSE51 profile for Minimal Real-time Systems and supports:

- Multi-threading
- Standard device I/O

- Standard file I/O (PSE52)
- Standard device Shared Memory routines
- File system interface for accessing sockets (PSE53)
- File system interface for accessing shared memory objects
- Basic support for message passing (PSE52)
- Basic support for signals
- FAT File System

> Some features have limited functionality for devices without peripheral support.

For more information see the *TASKING POSIX Implementation*.

## 7.2. STM32 Repository

The STM32 repository contains all STM32 specific content, like peripherals, low-level drivers and library wrappers. Support is included for the following device families:

- STMicroelectronics STM32F0xx
- STMicroelectronics STM32F10x
- STMicroelectronics STM32F2xx
- STMicroelectronics STM32F30x
- STMicroelectronics STM32F37x
- STMicroelectronics STM32F4xx (no FP support)
- STMicroelectronics STM32L1xx

This repository contains software/firmware from STMicroelectronics which is licensed under MCD-ST Liberty SW License Agreement V2. You may not use this package except in compliance with this License. You may obtain a copy of the License at:

http://www.st.com/software_license_agreement_liberty_v2 (PDF)

## 7.3. EFM32 Repository

The EFM32 repository contains all Silicon Labs EFM32 specific content, like peripherals, low-level drivers and library wrappers. Support is included for the following device families:

- EFM32 Gecko
- EFM32 Giant Gecko
- EFM32 Leopard Gecko
- EFM32 Tiny Gecko
- EFM32 Wonder Gecko
- EFM32 Zero Gecko

This repository contains software/firmware from Silicon Laboratories which is licensed. See the top of each file for detailed information. Basically you are free to use the Silicon Labs code for any project using Silicon Labs devices.

## 7.4. Generic Repository

Software services, drivers and peripherals in the generic repository are provided as Bonus Technology as described in the Altium EULA. Some third-party software is included which may require special license agreements. Check the generated code for the exact license usage conditions.

### Software services

The following table contains an overview of the software services present in the generic repository.

| Software Service | Description |
|---|---|
| AES | Generic interface to Advanced Encryption Standard (AES) encryption modules. |
| Audio Service | Interface to audio controller services. |
| Bluetooth HID Service | Bluetooth Human Interface Device (HID) class. Interfaces to bluetooth service. |
| Bluetooth Service | Bluetooth interface. |
| Ethernet Protocol Services | Service for ethernet peripherals. |
| GPS-NMEA Services | GPS-NMEA decoder. |
| Graphics Services | Service for graphics routines, low-level drawing functionality. |
| GUI Services | The Altium Graphical User Interface (AGUI) contains components you can use to build a User Interface. |
| HyperText Transfer Protocol | HTTP interface. |
| Interrupt Management | Interrupt services. Used by interrupt-driven driver software. |

| Software Service | Description |
|---|---|
| JPEG Decoder | Basic implementation to decode a JPEG image. |
| JPEG Encoder | Basic implementation to encode a JPEG image. |
| JSON Parser | Interface to YAJL's JSON generation facilities. |
| Location Services | Generic interface to GPS-NMEA or Simcom AT GPS services. |
| Matrix SSL / TLS | SSL and TLS implementation designed for small footprint applications and devices. |
| Modem Services | Services to communicate with modems using Hayes AT commands. |
| Point-to-Point Protocol Services | LWIP PPP service. |
| Pointer Services | The pointer service keeps the current pointer position up-to-date depending on the input from mouse and/or touchscreen. |
| Simcom AT GPS Service | GPS receiver. |
| Software Timing Services | Timing and timer services. |
| TCP/IP Networking | AutoIP Automatic LinkLocal IP Configuration. |
| Touchscreen Services | Interface to touchscreen controller services. |
| USB Host Bluetooth HCI Service | Driver for USB Host Bluetooth HCI. |
| USB Host Hub Service | Driver for USB Host Hub. |
| USB Host Keyboard Service | Driver for USB Host Keyboard. |
| USB Host Mass Storage Service | Driver for USB Host MSD. |
| USB Host Mouse Service | Driver for USB Host Mouse. |
| USB Host Service | Generic USB Host service. |
| USB Host Video Service | Driver for USB Host UVC. |
| UUID Generator | This module partially implements RFC4122. |

## Generic drivers

Generic drivers provide a generic interface to a certain type of peripheral. If the application uses the generic interface, then the hardware specific driver can be changed without the need to modify the application. For example, you can have a generic ADC driver that interfaces with an ADC084S021 ADC driver (SPI).

| Driver | Description |
|---|---|
| 802.15.4 LR-WPAN | Generic low-rate wireless PAN driver. |
| AES Encryption/Decryption | Software implementation of the Advanced Encryption Standard (AES). |
| External Interrupt | Platform independent generic external interrupt support. |
| Generic ADC Driver | Generic interface for Analog-to-Digital Converters. |

| Driver | Description |
|---|---|
| Generic CAN Driver | Generic interface for Controller Area Networks. |
| Generic DAC Driver | Generic interface for Digital-to-Analog Converters. |
| Generic PWM Driver | Generic interface for Pulse-Width Modulation. |
| Generic RTC Driver | Generic interface for Real Time Clocks. |
| RAM Disk | Interface for the creation of memory based file systems. |

## External drivers

External drivers are drivers for off-chip peripherals. These off-chip peripherals are connected to the chip as an SPI slave device, an I2C slave device, or via some other interface such as GPIO or DMA. In the former two cases an on-chip SPI master peripheral or an on-chip I2C master peripheral is required. For example, the ADC084S021 ADC driver (SPI slave) requires the STM32 SPI driver.

The following tables contain an overview of the external drivers present in the generic repository. Some drivers appear in more than one table, which means that they can have more than one interface.

## External drivers - I2C

| Driver | Description |
|---|---|
| APDS9300 Light Photo Sensor Driver | External driver to turn power on or off of a light photo sensor. |
| BMP085 Pressure Sensor Driver | External driver to interface with the BMP085 Digital Pressure Sensor. |
| CS4322 Audio Codec Driver | Interface to I2C control port of CS4322 stereo audio CODEC. |
| LIS302DL MEMS Motion Sensor | Interface to LIS302DL MEMS motion sensor. |
| LM75 Temperature Sensor | Interface to STLM75 Digital Temperature Sensor & Thermal Watchdog. |
| M24Cxx Serial I2C Bus EEPROM Driver | Interface to I2C based M24Cxx serial flash memory device driver |
| MAX1037 ADC Driver | Interface to the MAX1037 AD converter device driver. |
| MAX5841 DAC Driver | Interface to the MAX5841 DA converter device driver. |
| MAX6966 LED Driver | Device driver for MAX6966 (10-Port Constant-Current LED Driver) peripheral. |
| MCP3021 A/D Converter | Device driver for MCP3021 A/D Converter. |
| SSD0300 I2C OLED Driver | Driver for SSD0300 OLED display over I2C. |
| STMPE811 Touchscreen Controller / GPIO Expander | Interface to STMPE811 touchscreen controller / GPIO expander. |
| TVP5150 Video Decoder Driver | Driver to access configuration registers of a TVP5150 video input decoder device over an I2C bus. |

## External drivers - SPI

| Driver | Description |
|---|---|
| ADC084S021 ADC Driver | Four-channel CMOS 8-bit analog-to-digital converter with a high-speed serial interface. |
| ADC104S021 ADC Driver | Four-channel CMOS 10-bit analog-to-digital converter with a high-speed serial interface. |
| APDS9300 Light Photo Sensor Driver | External driver to turn power on or off of a light photo sensor. |
| AT86RF231 Radio Device Driver | Interface to the device driver for AT86RF231 802.15.4 packet radio. |
| CS4270 Audio Codec Driver | Interface to SPI control port of CS4270 stereo audio CODEC. |
| DAC084S085 DAC Driver | Interface to the DAC084S085 DA converter device driver. |
| DAC104S085 DAC Driver | Interface to the DAC104S085 DA converter device driver. |
| DS1391 Real Time Clock Driver | Device driver for DS1391 (Real Time Clock) peripheral. |
| ENC28J60 Driver | Interface to ENC28J60 ethernet controller device driver. |
| ICS307 Clock Generator Driver | External driver for ICS307-02 serially programmable clock source. |
| ILI9320 LCD Module Driver (SPI) | Device driver for an ILI9320 based LCD that is connected through SPI. |
| ILI9341 LCD Module Driver | Device driver for the ILI9341 Display Controller. |
| M25Px0 Serial Flash Memory Driver | Interface to SPI based M25Px0 serial flash memory device driver. |
| MAX1104 Audio Codec Driver | The MAX1104 CODEC provides both an 8-bit analog-to-digital converter (ADC) and an 8-bit digital-to-analog converter (DAC). |
| MAX1229 12x12 ADC Driver | Device driver for MAX1229 (Real Time Clock) peripheral. |
| PCF2123 Real Time Clock Driver | Device driver for PCF2123 (Real Time Clock) peripheral. |
| SD Card Reader Driver | Driver to access SD card readers through an SPI controller. |
| SPFD5408 LCD Module Driver (SPI) | Device driver for SPFD5408 based LCDs through SPI connections. |
| SSD1329 SPI OLED Driver | Driver for SSD1329 OLED display over SPI. |

## External drivers - other

| Driver | Description |
|---|---|
| APDS9700 Proximity Sensor Driver | External driver to interface with a proximity sensor. |
| AT86RF231 Radio Device Driver | Interface to the device driver for AT86RF231 802.15.4 packet radio. |
| DWC USB OTG Driver | DesignWare Core portability library, low-level OS wrapper functions. |

| Driver | Description |
|---|---|
| HIH5030 Humidity Sensor Driver | External driver to interface with a humidity sensor. |
| ILI9320 LCD Module Driver (parallel) | Device driver for an ILI9320 based LCD that is connected to the parallel interface. |
| ILI9341 LCD Module Driver | Device driver for the ILI9341 Display Controller. |
| M25Px0 Flash Memory Block I/O | Interface to M25Px0 parallel flash memory device driver. |
| SPFD5408 LCD Module Driver (parallel) | Device driver for SPFD5408 based LCDs through parallel connections. |
| SSD1329 SPI OLED Driver | Driver for SSD1329 OLED display over SPI. |
| ST7637_PAR LCD Driver | Driver for ST7637 LCD display with parallel interface. |

# Chapter 8. Glossary

The following terminology is used to describe the TASKING Software Platform concepts (in alphabetical order).

## Software Platform

The source code generated by the Software Platform Builder. The source code becomes part of your project. A graphical representation of the Software Platform is present in the Software Platform Builder.

## Software Platform Builder

The graphical editor where you can edit a Software Platform document and where you can generate the source code for your Software Platform. The editor shows a graphical representation of your Software Platform.

## Software Platform document

A file with extension `.swpxmi` which is part of your project. It contains the configuration of your Software Platform and is managed by the Software Platform Builder.

## Software Platform plug-in or module

Software Platform definition files, source code and other resources. Plug-ins are bundled in a Software Platform repository.

## Software Platform repository

The collection of Software Platform plug-ins. It provides the contents for a generated Software Platform.

## TASKING Software Platform

The main software framework delivered by Altium, consisting of Software Platform repositories and the Software Platform Builder.