NXP Semiconductors

Creating an USB Host Project (MSC + fatfs) with KSDK 1.3 and Processor Expert support in KDS

By: Technical Information Center

Kinetis Software Development Kit (KSDK) offers an USB stack for Device, Host and/or OTG functionality. This stack is easy to use and speeds up designs for users, however, application sometimes goes beyond USB functionality and requires adding more drivers and/or functions that could be added easily by using Processor Expert tool.

Currently, Processor Expert (PEx) does not have any USB Host component that could aid user to design his/her USB Host application like it is done in USB Device mode. This guide is focused to illustrate the steps needed to create a basic USB Host Project with Mass Storage Class (MSC) support and fatfs File System with KSDK 1.3 and Processor Expert Support. Furthermore, it can be used as guidance for users in order to create an USB Host application with different class support.

This guide was focused on enabling USB Host MSC and fatfs on the FRDM-K64F board, however, it is not limited to use only this MCU and/or this specific class.

1 Requirements.

1.1 Install newest KDS version (Kinetis Design Studio 3.2.0), you can download from www.freescale.com/kds

1.2 Install KSDK 1.3.0 (Kinetis Software Development Kit), you can download from www.freescale.com/ksdk

1.3 Install ***'KSDK_1.3.0_Eclipse_Update'*** you can find the update in ***'<KSDK_1_3_PATH>\tools\eclipse_update'***. The instruction to make the updates are described in chapter 2 of ***'<KSDK_1_3_PATH>\doc\rtos\mqx\MQX RTOS IDE Guides\MQX-KSDK-KDS-Getting-Started.pdf'***

2 Create a New Kinetis Project

2.1 Open Kinetis Design Studio (KDS), create a new Kinetis project for your target and give it a name.

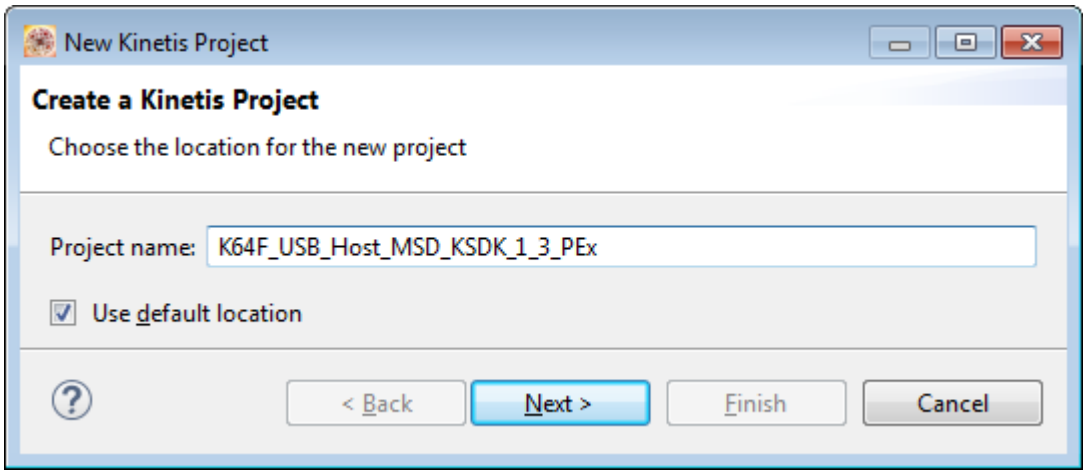

Figure 1 - Creating a New Kinetis Project

2.2 Select your target in the next window, in this guide MK64FN1M0xxx12 MCU will be used (This MCU is presented in FRDM-K64F board).

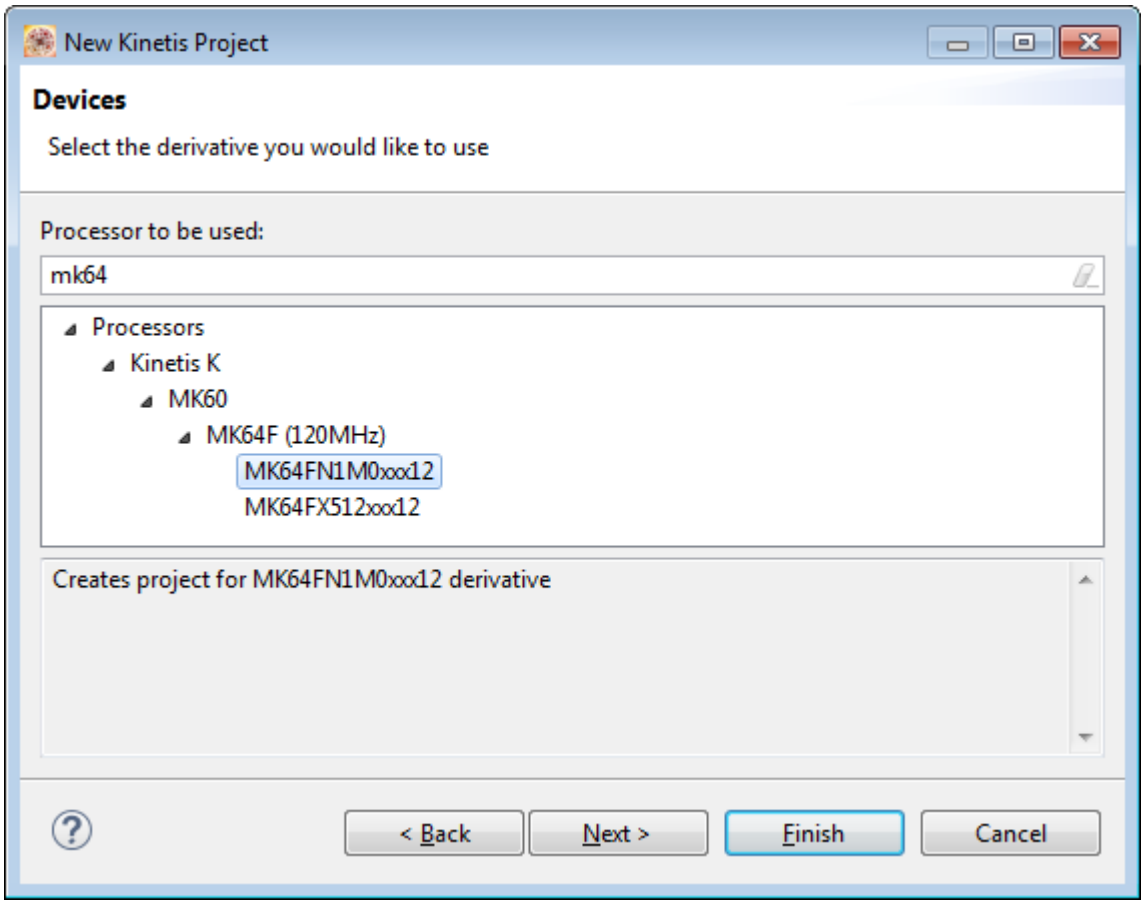

Figure 2 - Select MCU

2.3 Select KSDK 1.3.0 path and add Processor Expert support for current project. Then click on ***Finish*** button

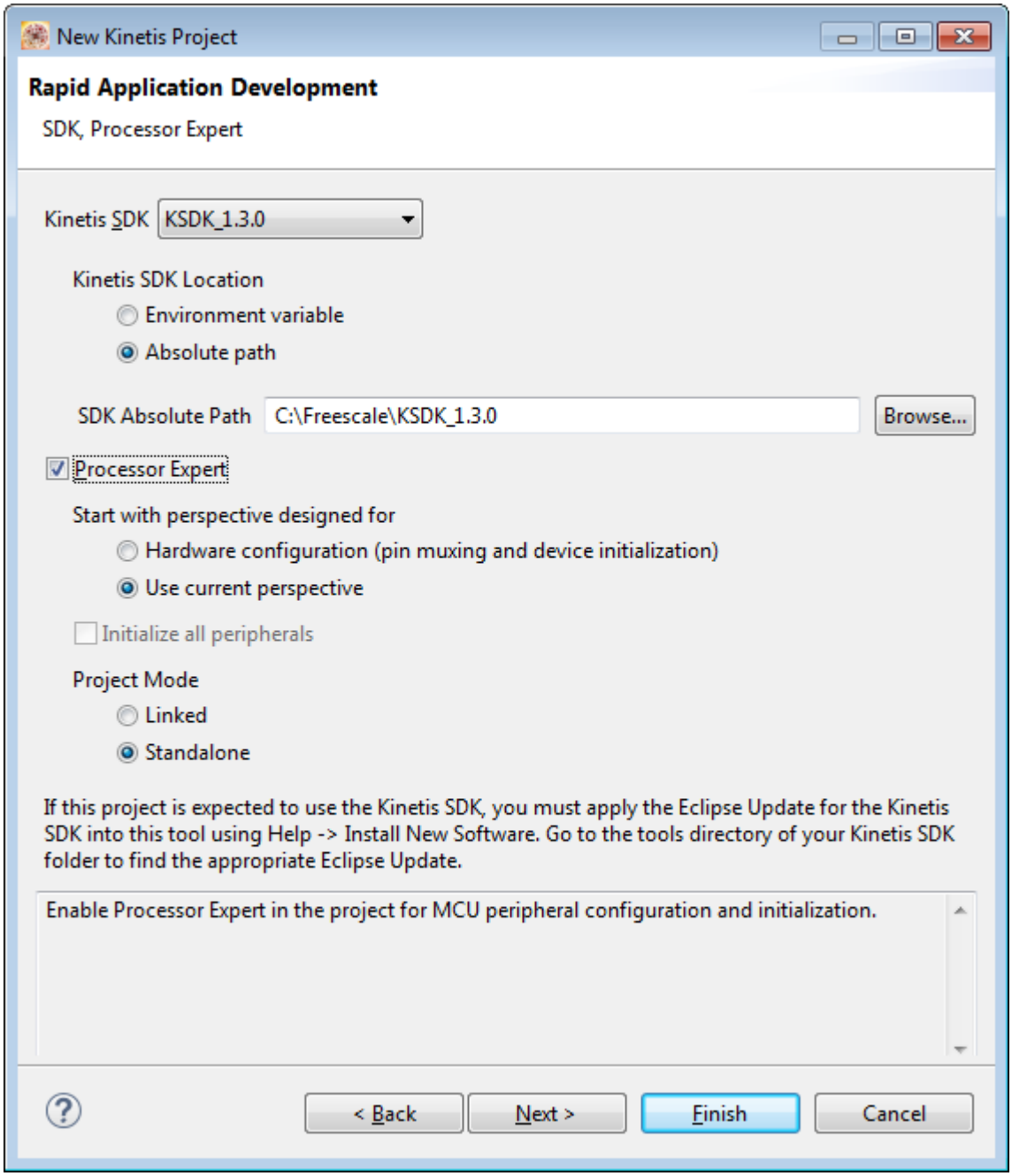

Figure 3 - Add Processor Expert support

3 Adjust clock settings

USB module requires a 48MHz clock and Core clock must be set minimum to 20 MHz for optimal operation. Through this chapter, a new clock configuration will be set to achieve these requirements.

3.1 In ***fsl_clock_manager*** component, go to *Clock configurations* tab and add a new configuration. Enter a description to specify that this configuration is used to set proper USB clocks settings.

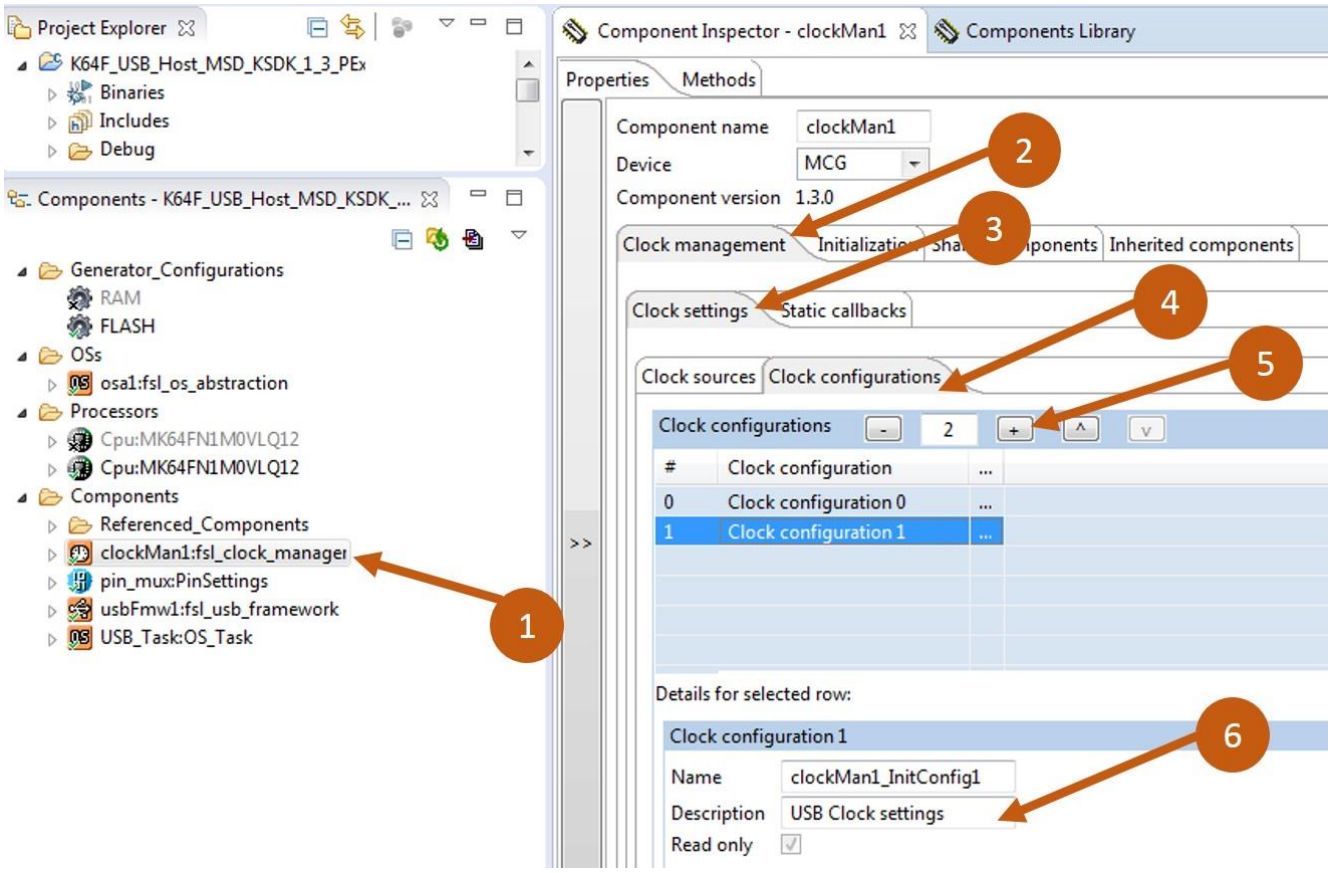

Figure 4 - Add a new clock configuration

3.2 Configure this clock configuration according to used hardware. In this case FRDM-K64F board is used, so an external 50 MHz reference clock feeds the MCU. Go to *Clock sources* tab and configure System oscillator 0.

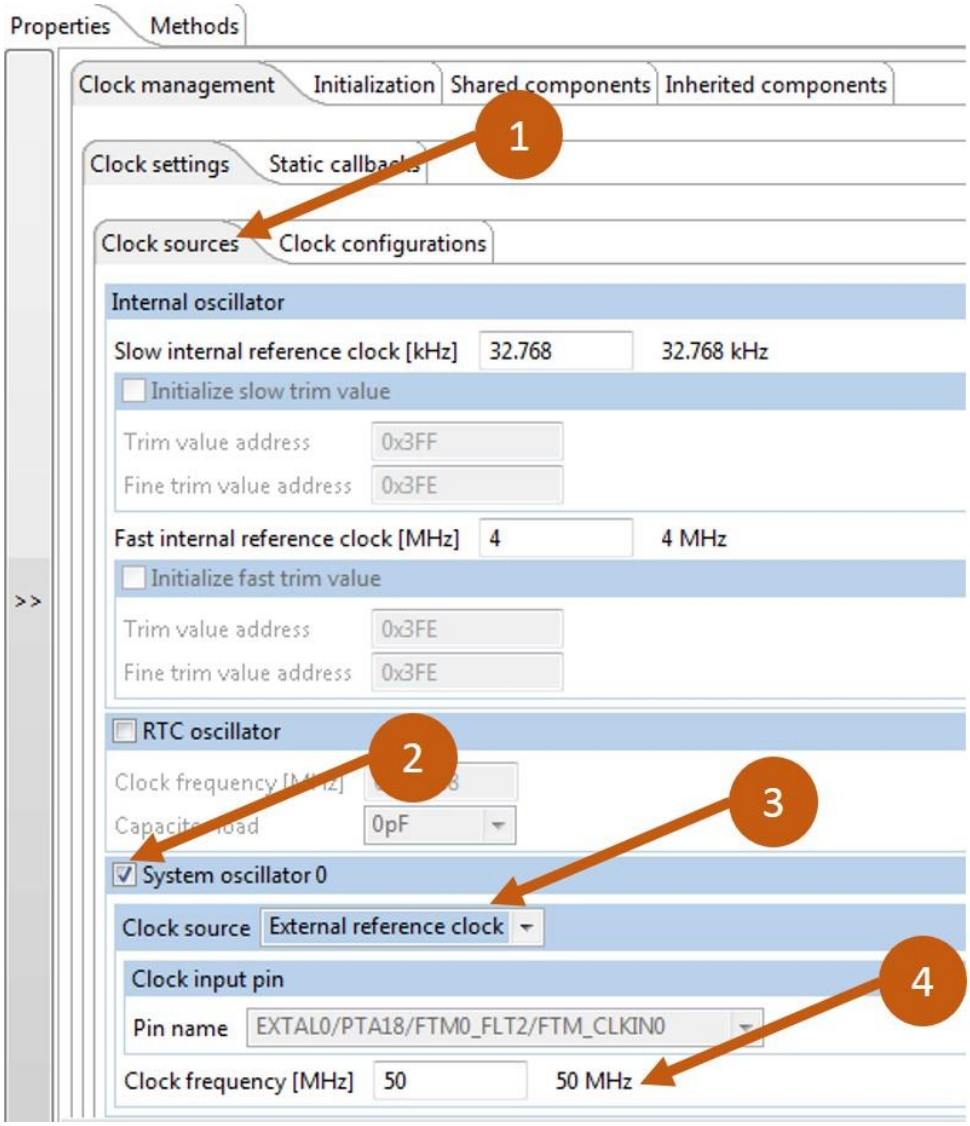

Figure 5 - Configure System Oscillator 0

3.3 In *Clock configurations* tab, go to *MCG Settings* tab for new configuration and set MCG mode to PEE besides setting PLL output frequency to its maximum value: 120MHz.

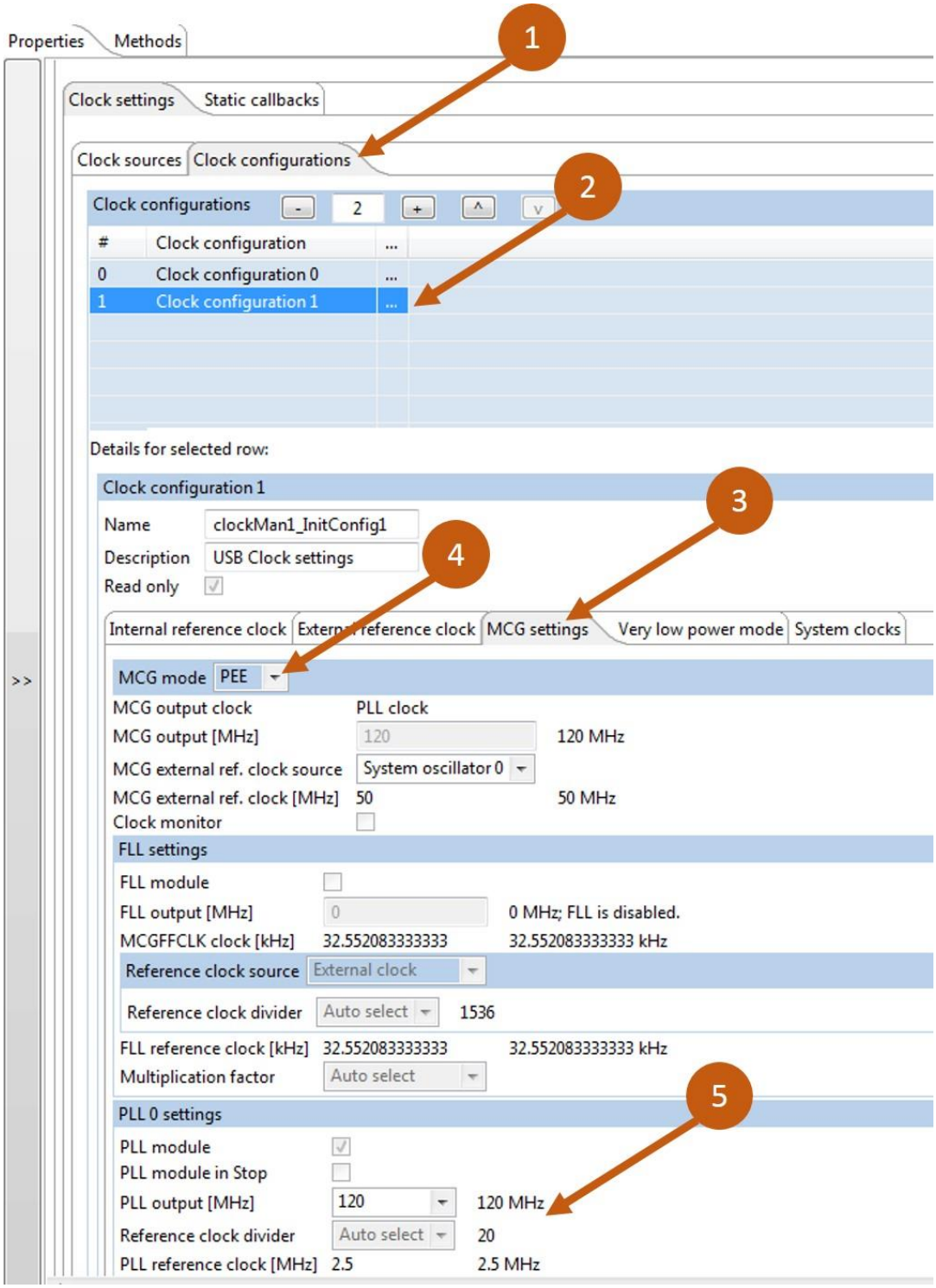

Figure 6 - Set PEE mode and adjust PLL output frequency

3.4 In *System clocks* tab, adjust Core, Bus, External Bus and Flash clocks just as shown below. Check the USB clock settings and validate that USB clock is 48 MHz.

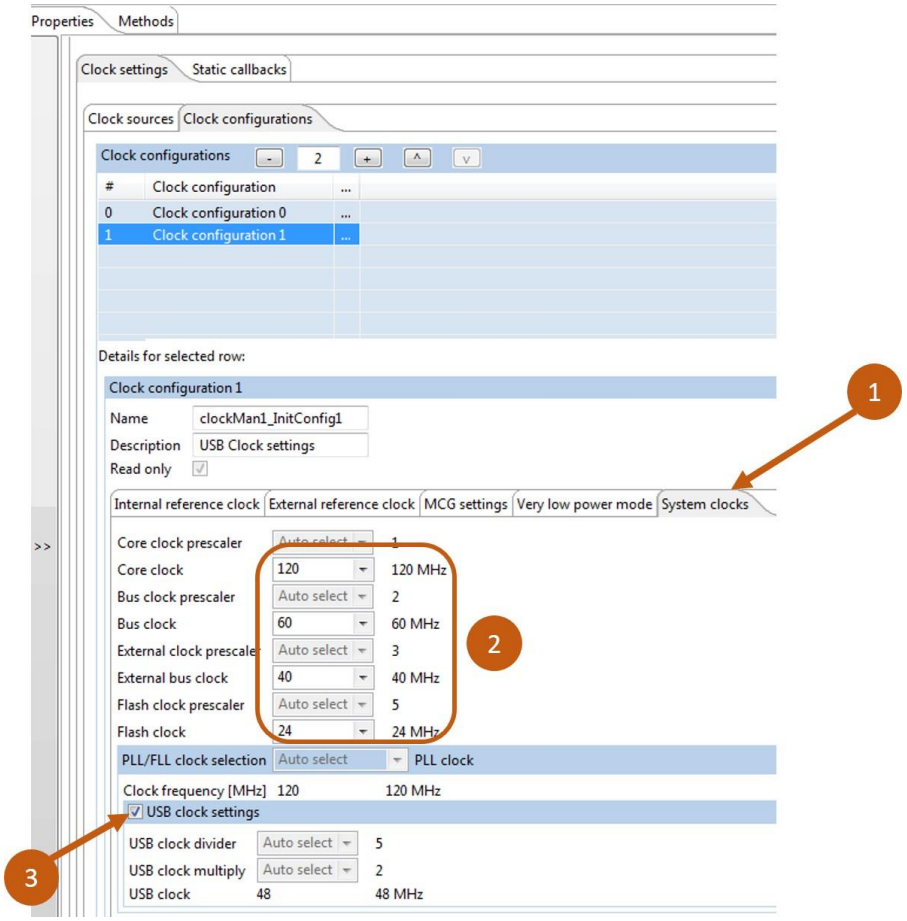

Figure 7 - Adjust system clocks and enable USB clock

3.5 Finally, go to *Initialization* tab and select this clock configuration for initialization procedure. Save changes for current component.

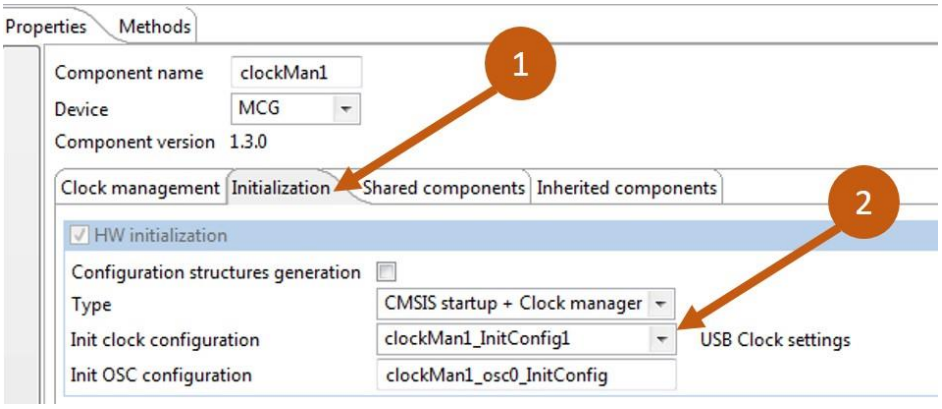

Figure 8 - Use USB clock configuration for initialization procedure

4 USB Component

Now that clocks for USB module and MCU core are set according specifications, it is time to add USB component and configure it.

4.1 Add ***fsl_usb_framework*** component. You will be asked to add ***Init_FMC*** and ***fsl_debug_console*** components as Referenced components, just Select *OK* in prompt windows.

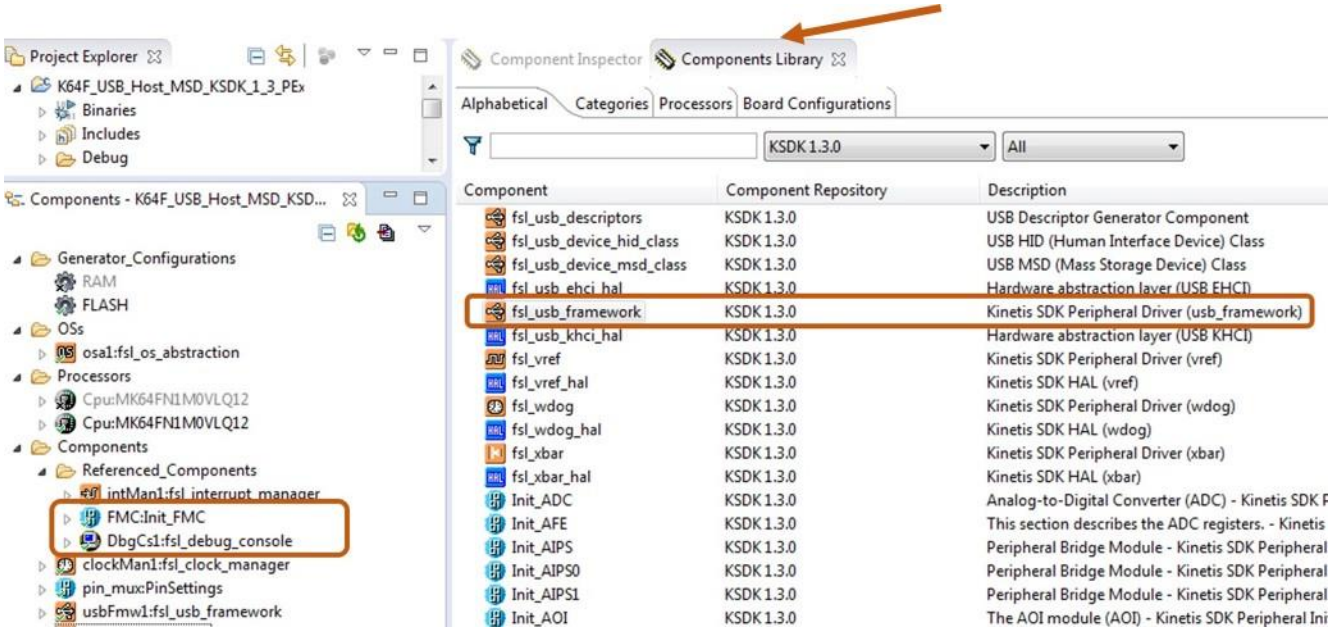

Figure 9 - Adding fsl_usb_framework component

Note: If you have installed other KSDK versions than 1.3, be sure to select the right component version for FMC and debug console.

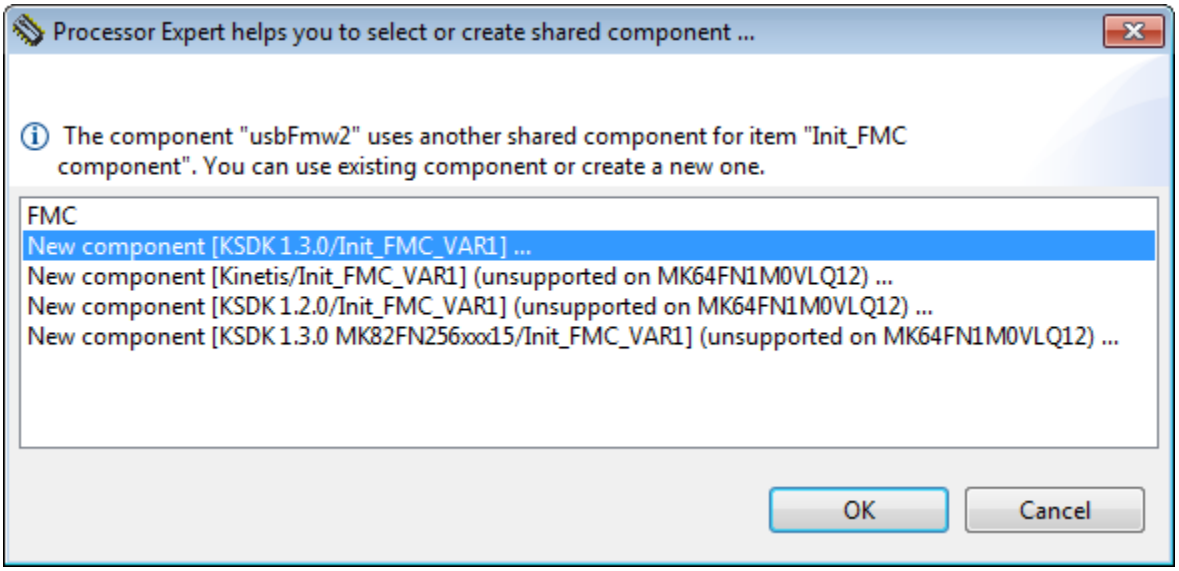

Figure 10 - Selecting Referenced_components

4.2 For ***fsl_debug_console*** be aware to select desired baud rate and pins that will be used in your hardware. For FRDM-K64F, default console is using UART0 and pins are PTB16 and PTB17.

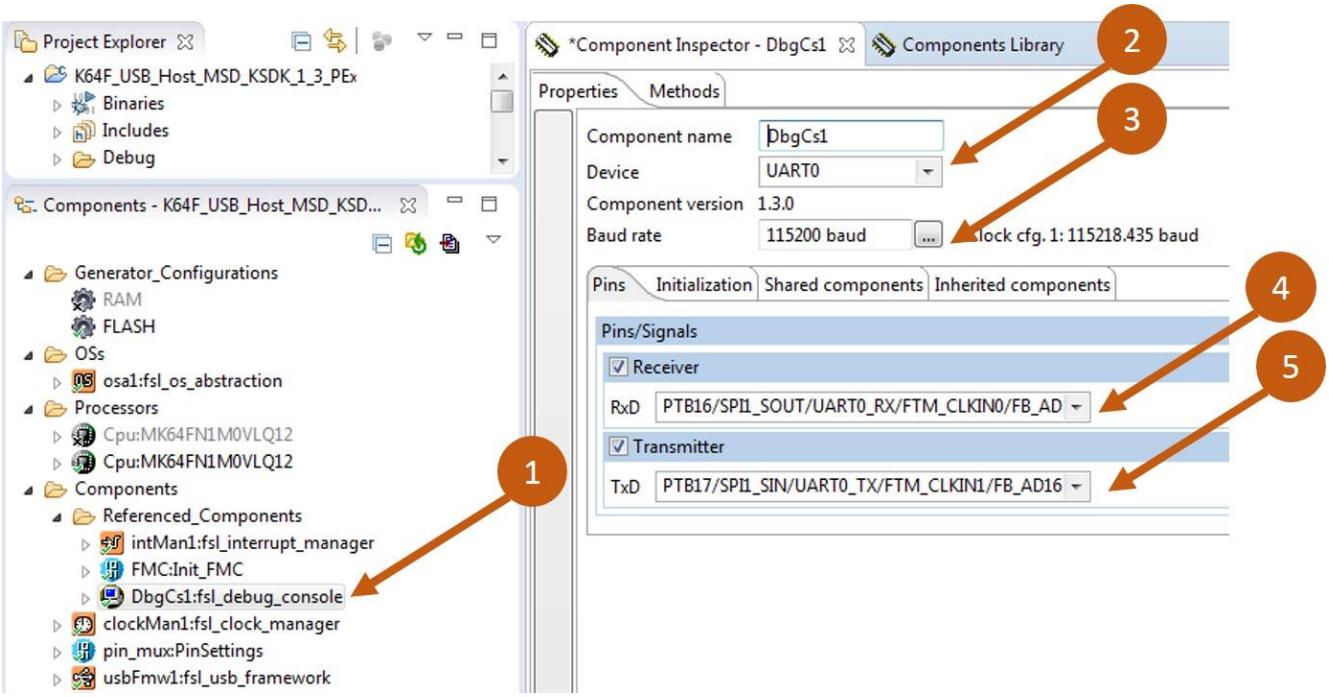

Figure 11 - fsl_debug_console configuration

4.3 For ***Init_FMC*** component, be sure that Master 4 (USB OTG) has Read only access protection.

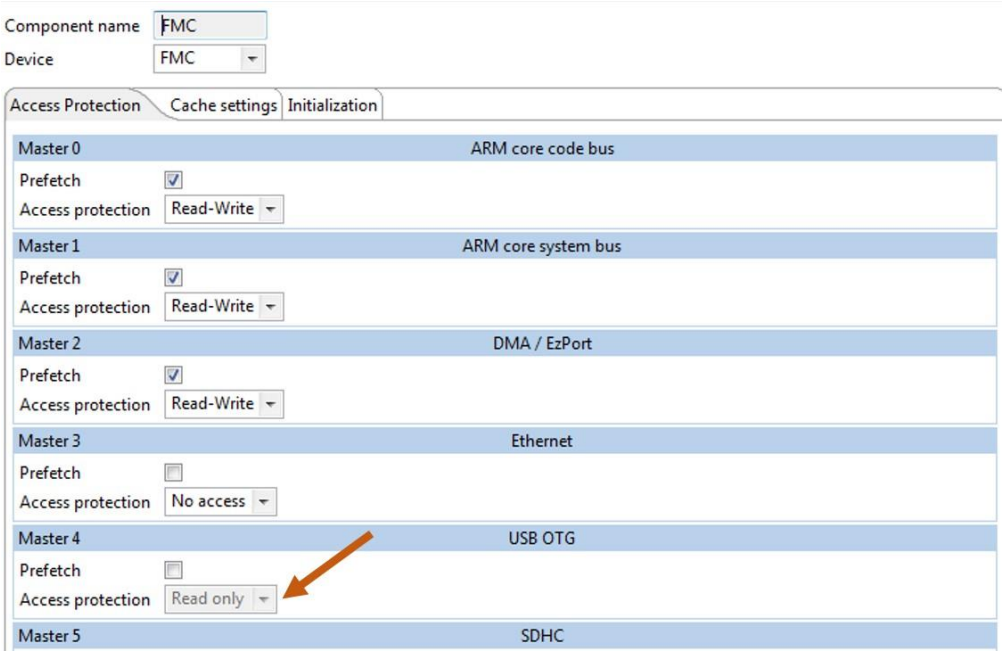

Figure 12 - FMC configuration

4.4 In ***fsl_usb_framework*** component, change Mode to HOST

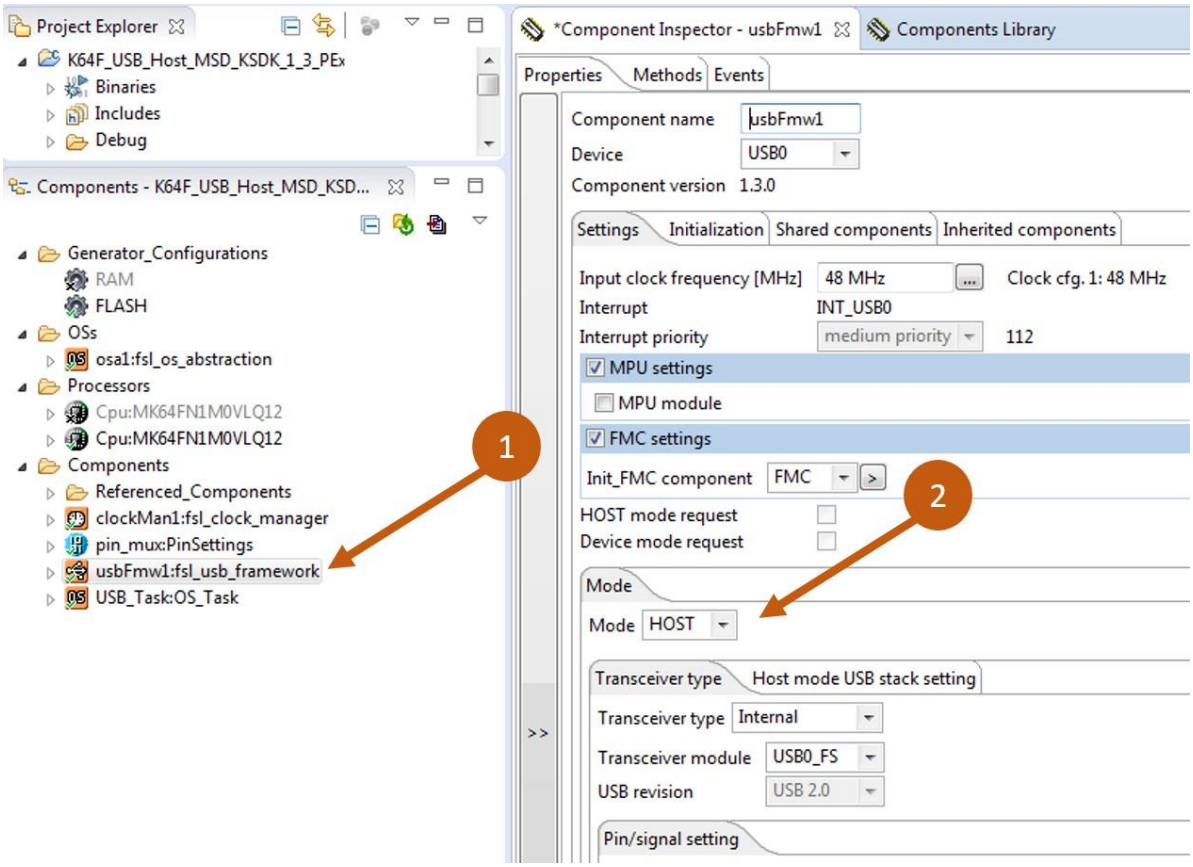

Figure 13 - USB Framework mode

4.5 In *Host Mode USB stack setting* tab, check the USBCFG_HOST_MSC and USBCFG_HOST_HUB class drivers.

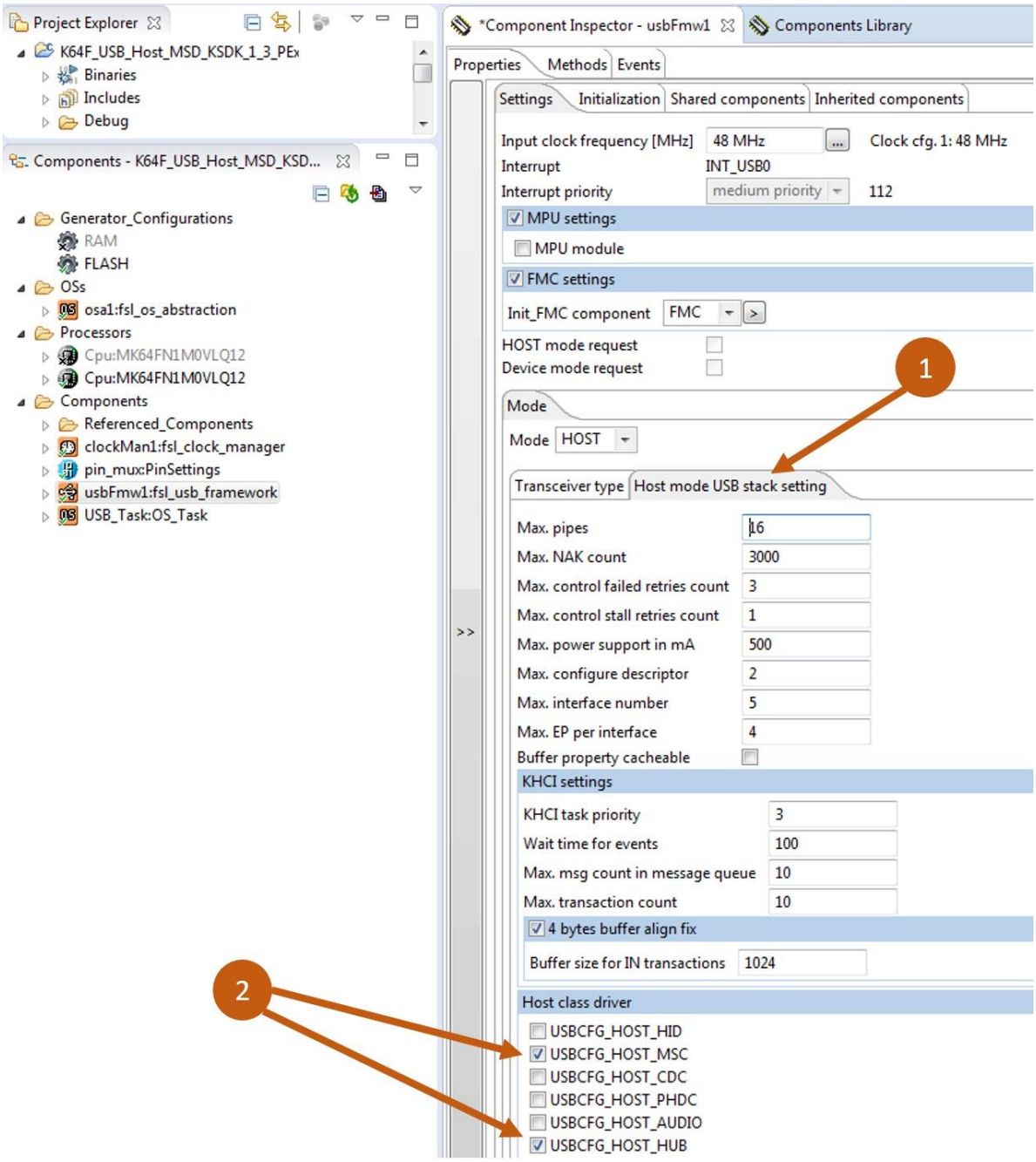

Figure 14 - Add class driver support for MSC and HUB

It is now necessary to include Driver information table for those devices that will be supported. This table includes sub-class and protocol information for supported devices.

4.6 Go to *Initialization* tab and uncheck the Device Mode field to clear the error, then, check the *Host mode* and fill the Device driver table as shown below:

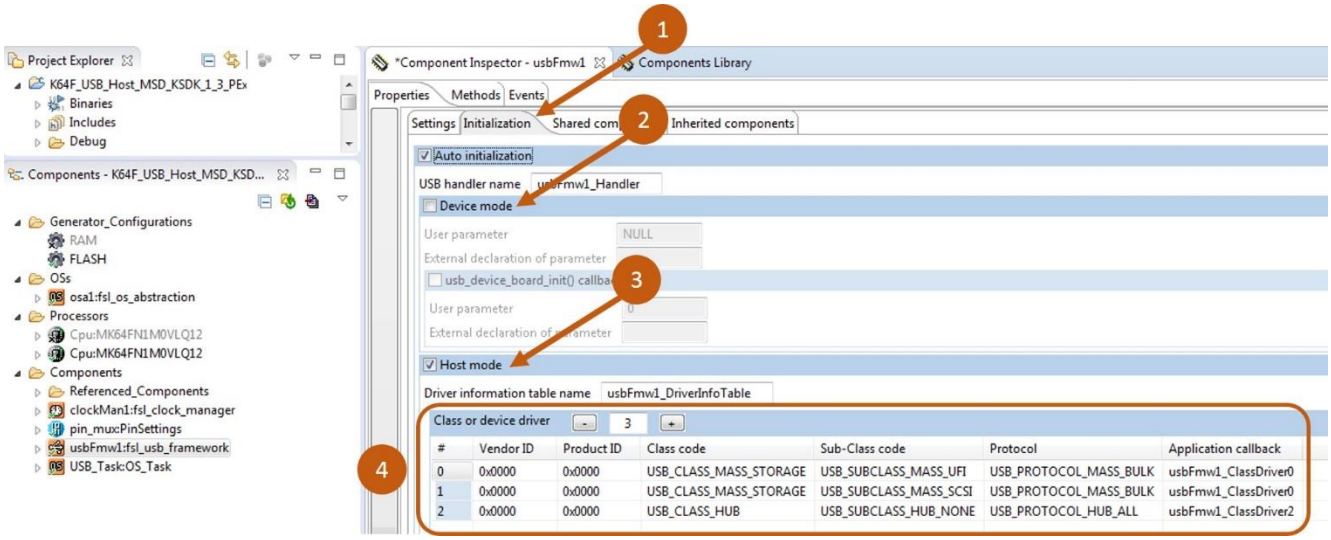

Figure 15 - Driver Info Table

Note: Please notice that Application callback (usbFmw1_ClassDriver0 in section 4 from previous image) is used for both Mass Storage Class drivers.

4.7 As both Mass Storage Class drivers are using the same application callback (usbFmw1_ClassDriver0), go to *Events* tab and select *do not generate code* for class_driver_callback1 in order to remove the error in ***fsl_usb_framework*** component.

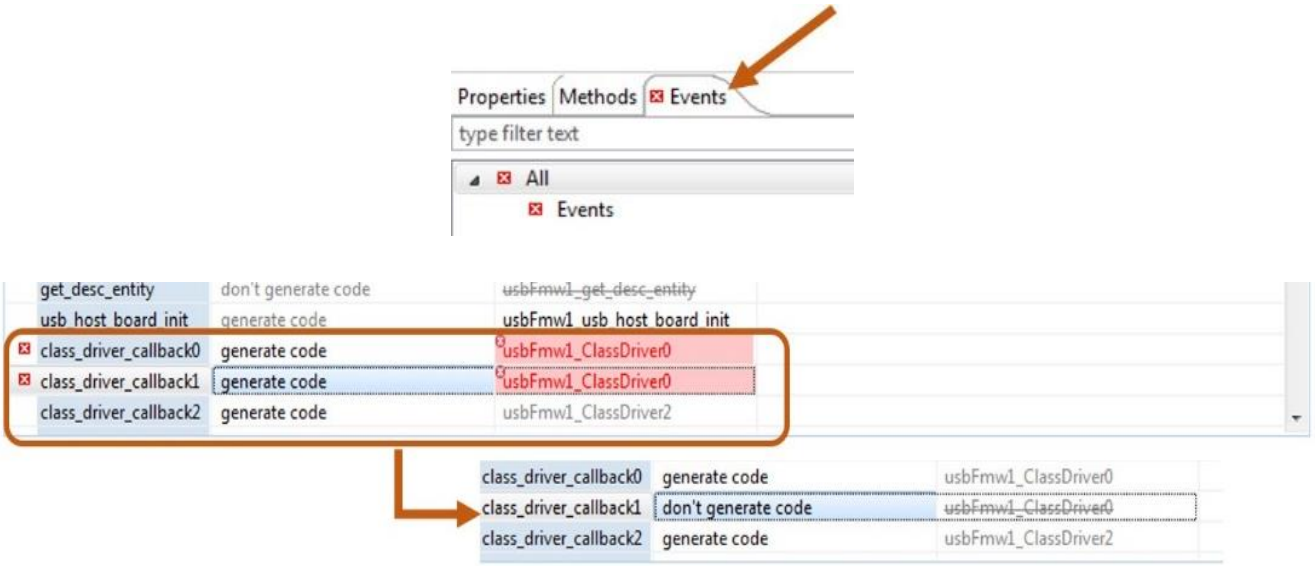

Figure 16 - Class Driver Callbacks generation

4.8 Save changes for ***fsl_usb_framework*** component.

5 USB task

In order to handle USB events like attachments, detachments, opening interfaces and more, a new task will be created.

5.1 Add ***OS_Task*** component and configure the new task (rename it).

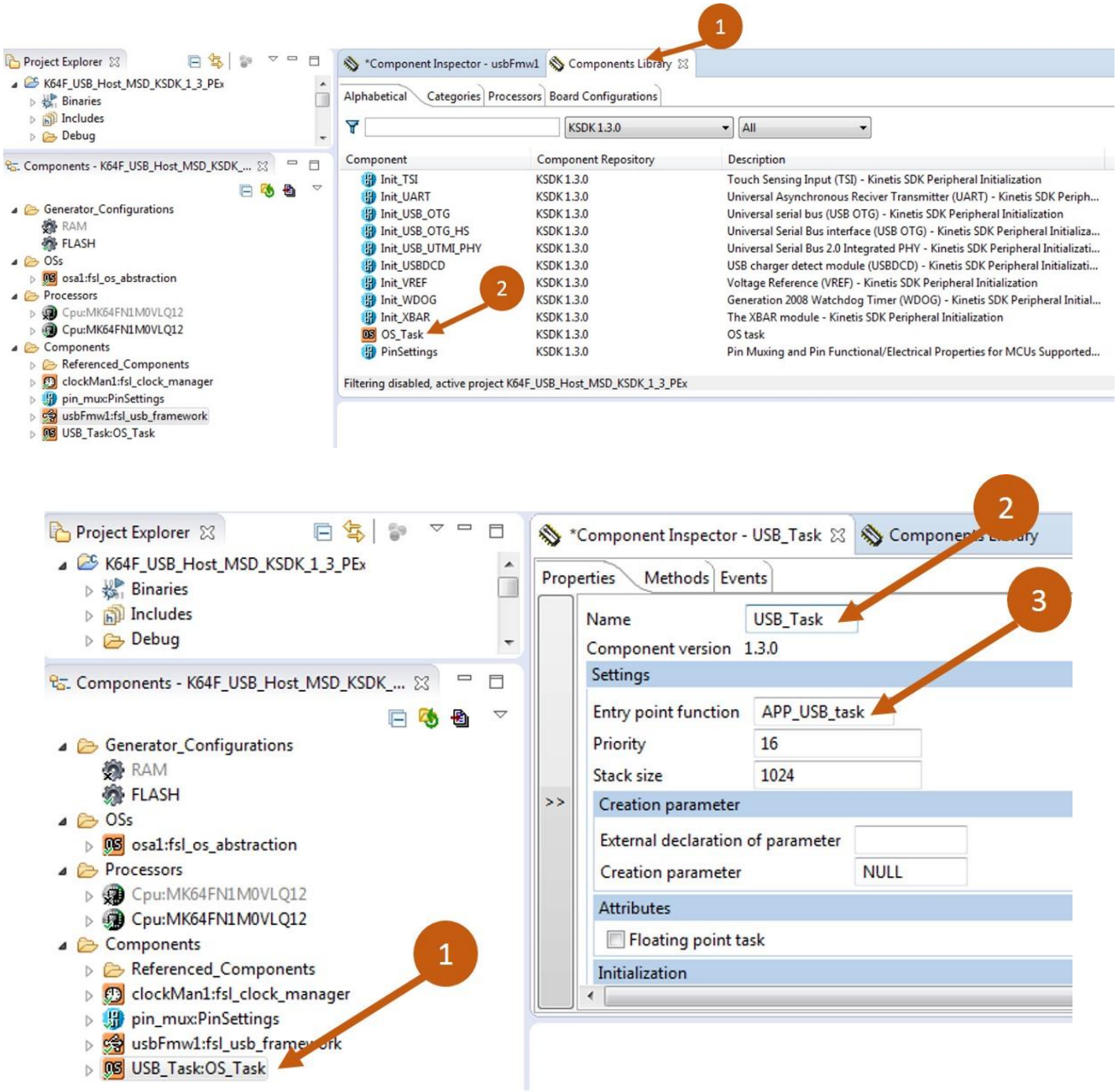

Figure 17 - USB task configuration

5.2 Lastly, save changes and generate code.

6 Configure USB Host stack

Until now, Processor Expert has created code for USB Host, however, Mass Storage Class and HUB files are still missing.

6.1 In ***SDK/usb/usb_core/host/sources/classes*** path create two new folders named ***msc*** and ***common*** just as it is shown below:

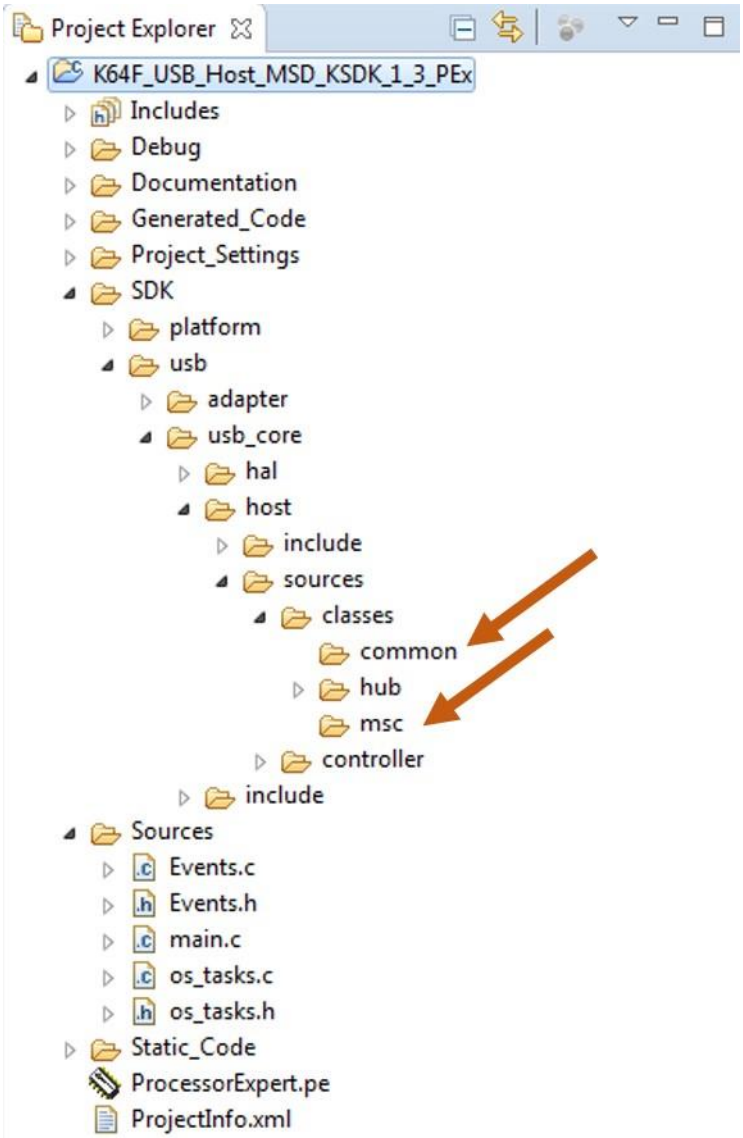

Figure 18 - Create MSC and Common folders

6.2 Inside these folders, copy files from *<KSDK_1_3_PATH>\usb\usb_core\host\sources\classes\msd* and *<KSDK_1_3_PATH>\usb\usb_core\host\sources\classes\common*. Also, add missing HUB files that are

located at <*KSDK_1_3_PATH*>*usb**usb_core**host**sources**classes**hub* inside hub folder. Next image shows the final structure for these folders.

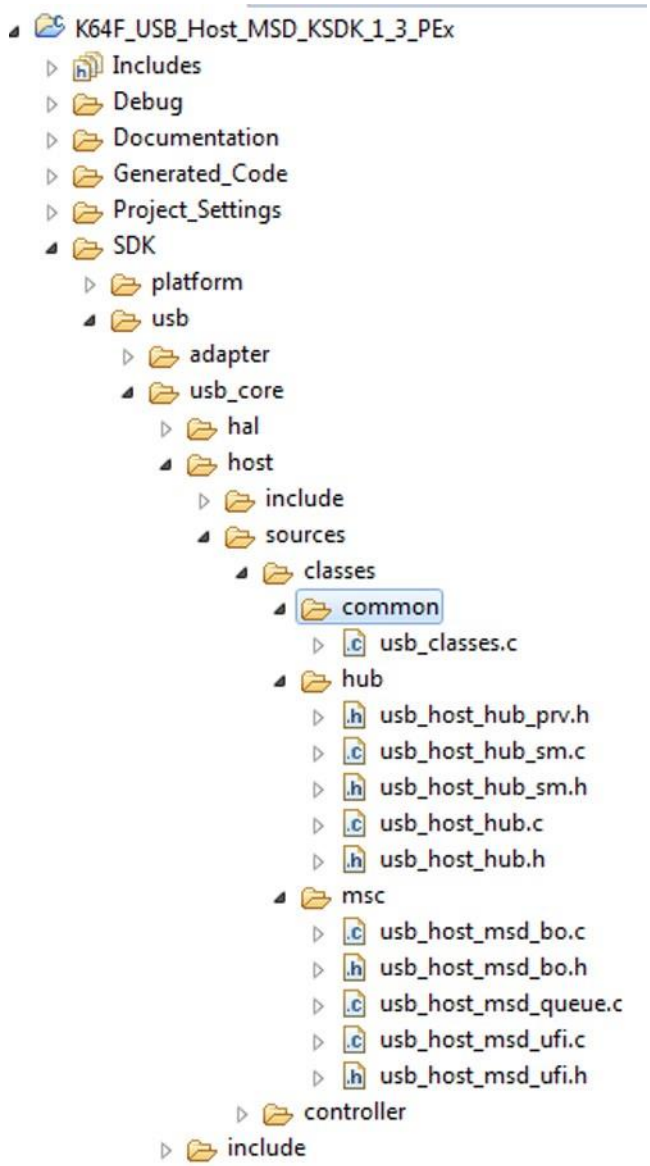

Figure 19 - USB Host Classes structure

6.3 Go to project's properties and add next path in C / C++ Build > Settings > Cross ARM C Compiler > Includes:

${ProjDirPath}/SDK/usb/usb_core/host/sources/classes/msc

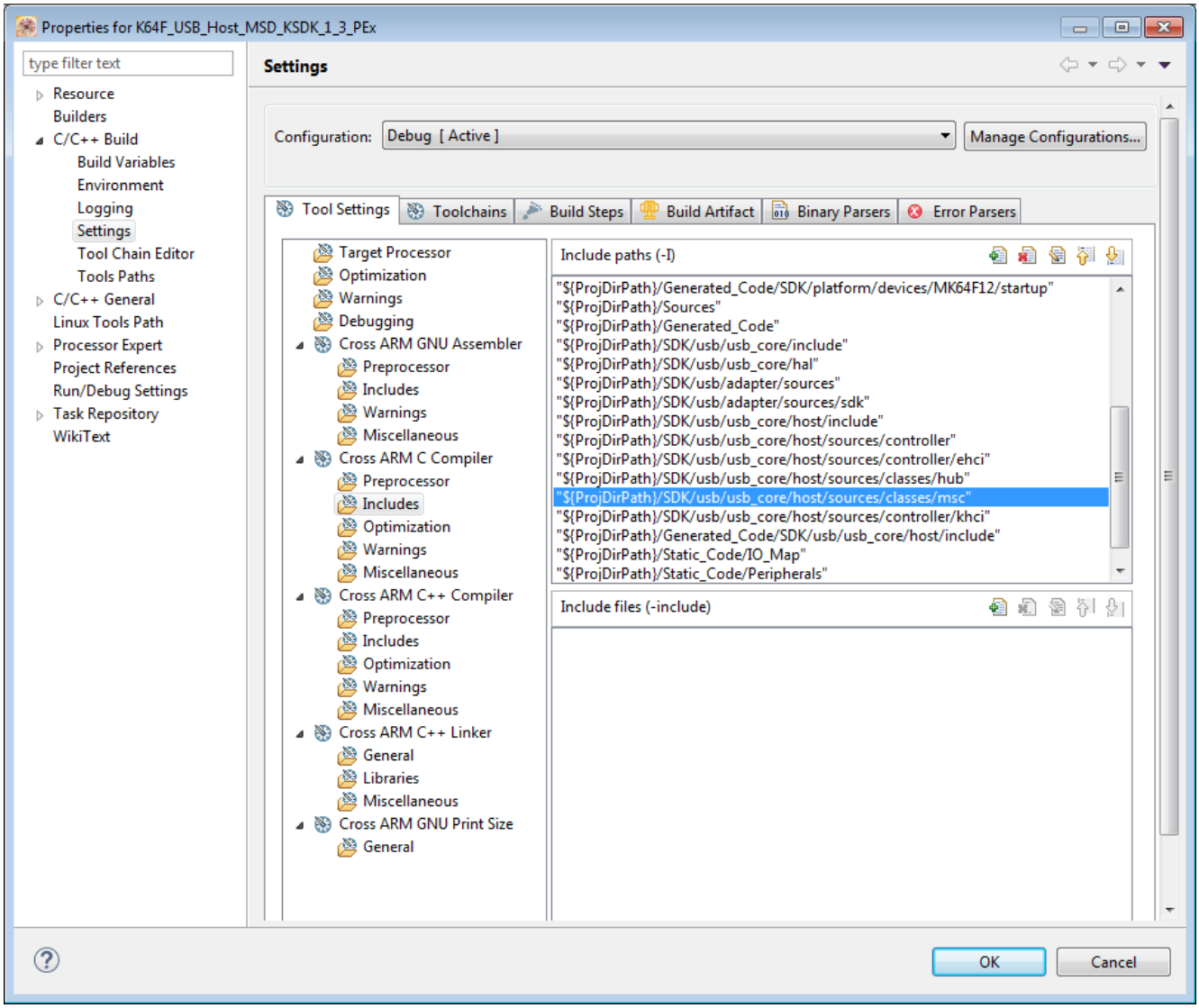

Figure 20 - Add path for MSC folder

6.4 Select ***Apply*** button and compile the project, at this point, there should be no errors.

7 Adding fatfs

By now, USB Host project supports Mass Storage Class, however, it is necessary to add File System support that will be mounted in attached devices.

7.1 Create a new folder named ***fatfs*** in the *Sources* folder.

7.2 In this new folder, add some of the files that are located at: *<KSDK_1_3_PATH>\middleware\filesystem\fatfs*, folder should look as follows:

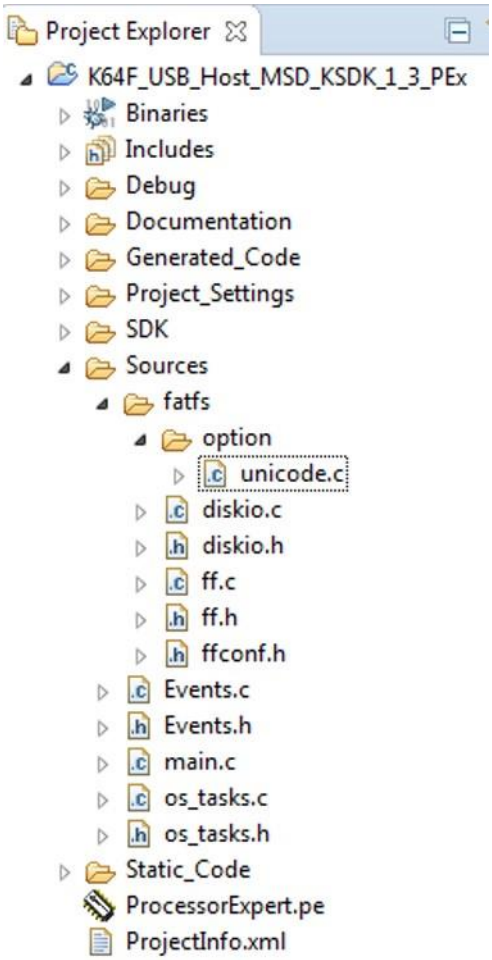

Figure 21 - fatfs structure

Note: Not all the files that are located at *<KSDK_1_3_PATH>\middleware\filesystem\fatfs* are required for this project. Copy only those that are shown in **Figure 21 - fatfs structure**.

7.3 It is necessary to add msd_diskio.h and msd_diskio.c files that are located inside ***fsl_usb_disk*** folder (*<KSDK_1_3_PATH>\middleware\filesystem\fatfs\fsl_usb_disk*). You can copy these files in Sources folder.

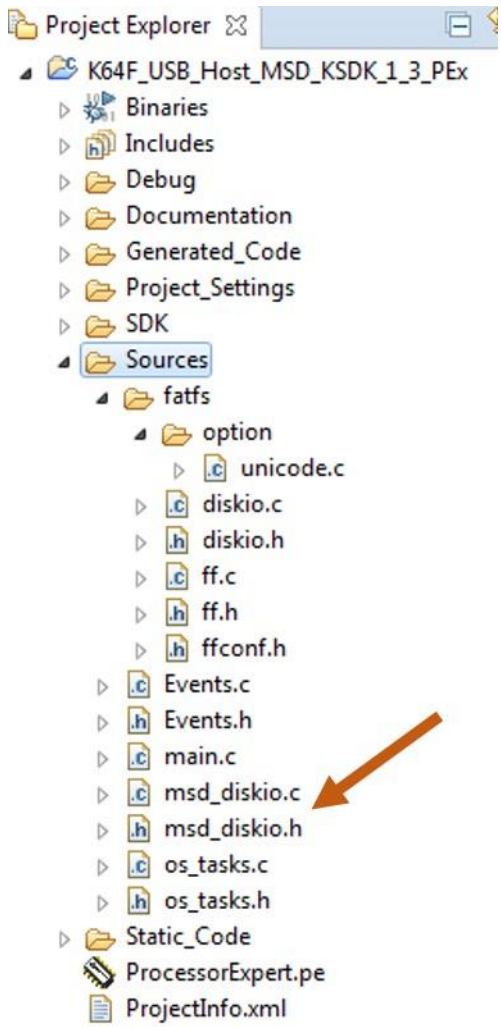

Figure 22 - Add msd_diskio files

7.4 In project's properties, add next path at C / C++ Build > Settings > Cross ARM C Compiler > Includes:

${ProjDirPath}/Sources/fatfs

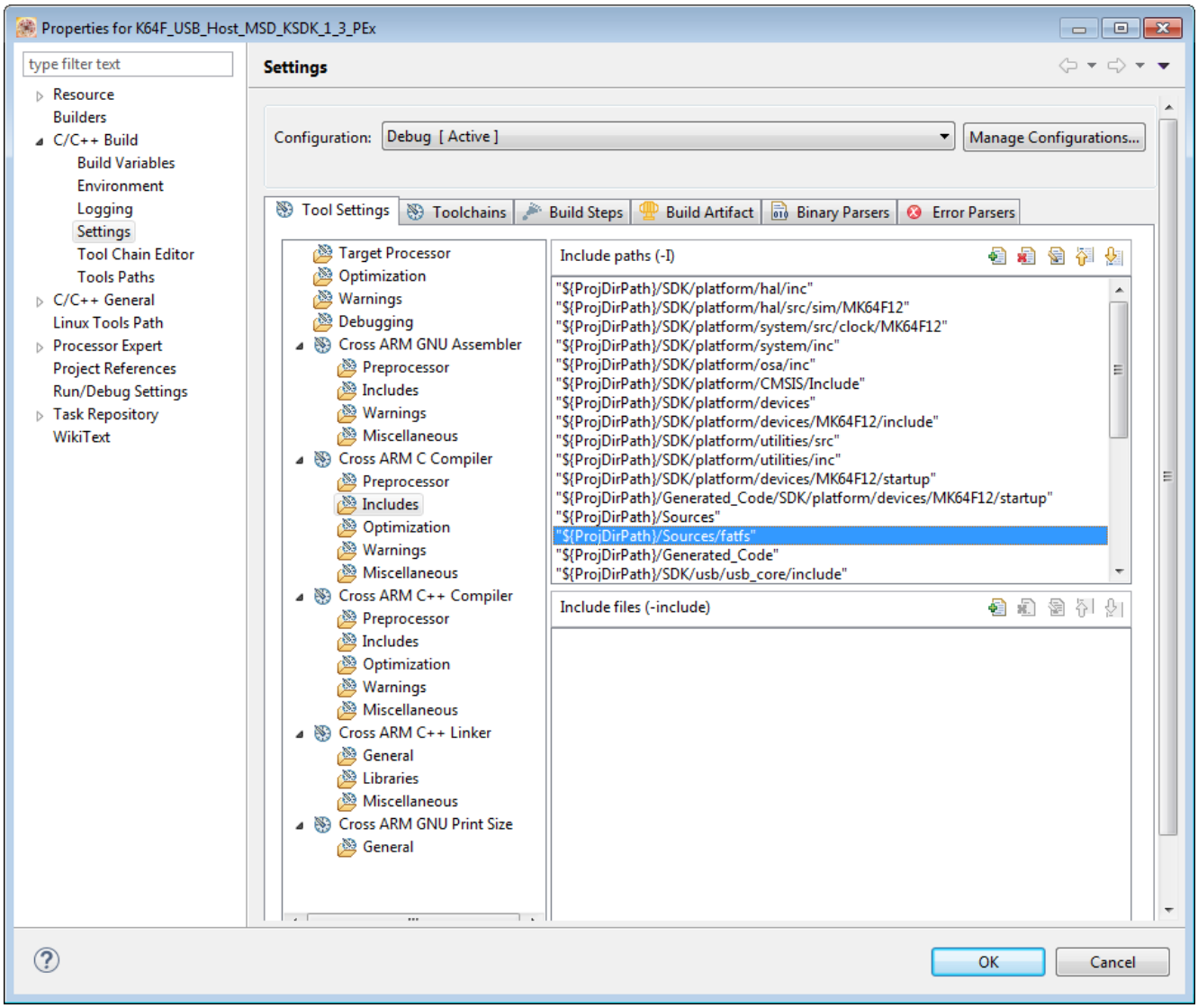

Figure 23 - Add fatfs folder path

7.5 In the diskio.c file there is the function disk_initialize(), which must be called by the application to mount the file system into the USB thumb. This function can initialize either USB and/or SD Card according to some macro definitions. In this case, it is necessary to define USB_DISK_ENABLE and set it to 1. Go to C / C++ Build > Settings > Cross ARM C Compiler > Preprocessor and add next macro:

USB_DISK_ENABLE=1

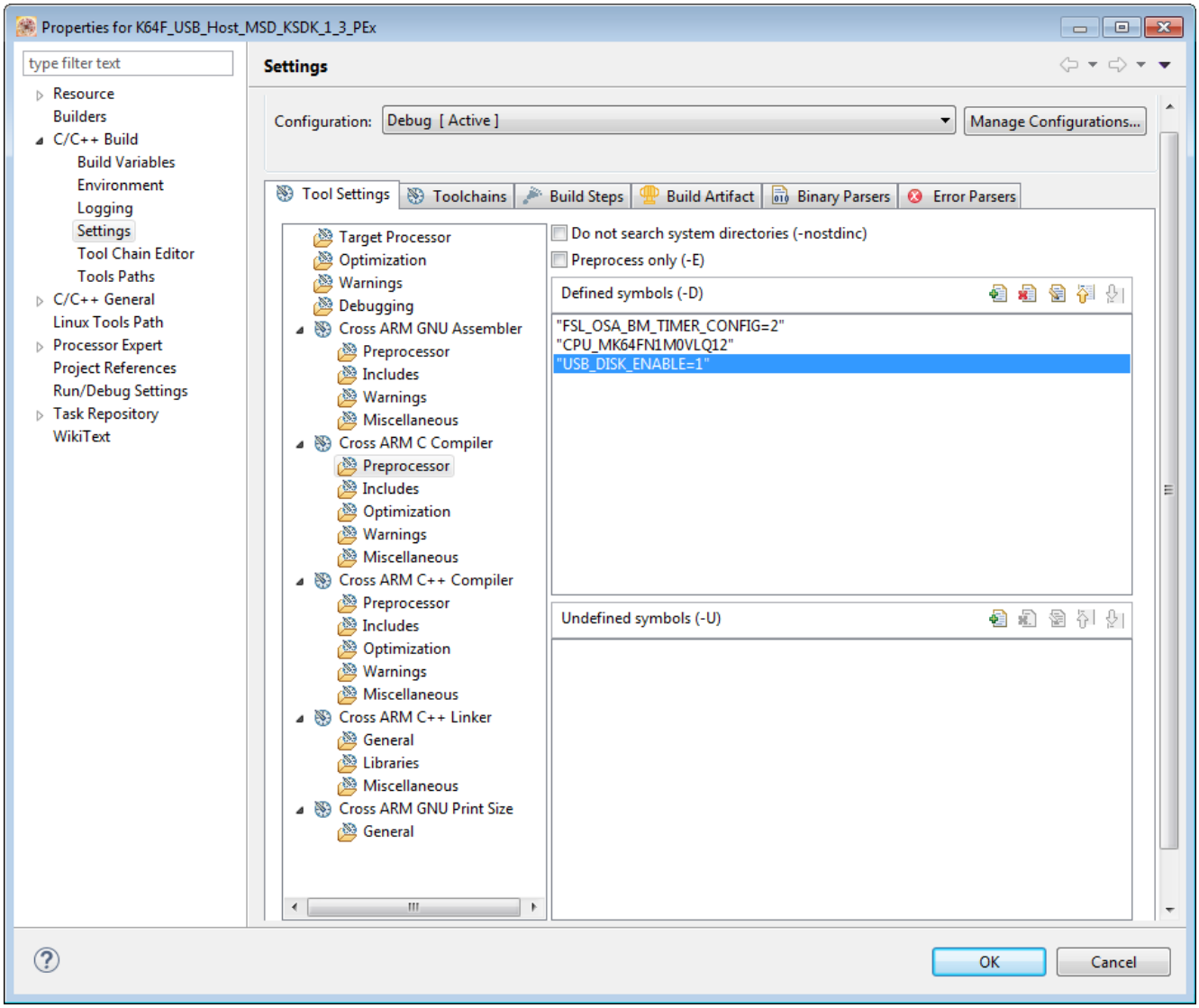

Figure 24 - Add USB_DISK_ENABLE macro

7.6 Project is now suitable to use USB Host with MSC and fatfs support in its application. It is time to create a basic application to test basic File System operation into an USB Mass Storage Device.

8 Create a basic application for Mass Storage Device and File System

8.1 In ***os_task.h*** file, include reference for msd_diskio.h file (**#include** "msd_diskio.h") and add the prototype for Mass Storage Driver callback:

```
usb_status usb_host_mass_device_event(usb_device_instance_handle dev_handle,
                                      usb_interface_descriptor_handle intf_handle,
                                      uint32_t event_code);
```

os_tasks.h

```
25  **  @addtogroup os_tasks_module os_tasks module documentation
26  **  @{
27  */
28
29  #ifndef __os_tasks_H
30  #define __os_tasks_H
31  /* MODULE os_tasks */
32
33  #include "fsl_device_registers.h"
34  #include "clockMan1.h"
35  #include "pin_mux.h"
36  #include "osa1.h"
37  #include "usbFmw1.h"
38  #include "DbgCs1.h"
39  #include "USB_Task.h"
40
41  #include "msd_diskio.h"
42
43  #ifdef __cplusplus
44  extern "C" {
45  #endif
46
47  /*
48  **  ===================================================================
49  **      Callback    : APP_USB_task
50  **      Description : Task function entry.
51  **      Parameters  :
52  **          task_init_data - OS task parameter
53  **      Returns : Nothing
54  **  ===================================================================
55  */
56  void APP_USB_task(os_task_param_t task_init_data);
57
58  usb_status usb_host_mass_device_event(usb_device_instance_handle dev_handle,
59                                        usb_interface_descriptor_handle intf_handle,
60                                        uint32_t event_code);
61  /* END os_tasks */
62
63  #ifdef __cplusplus
64  }  /* extern "C" */
65  #endif
66
67  #endif
```

Figure 25 - os_task.h file

8.2 In ***os_task.c*** add next definitions that will be used for usb_host_mass_device_event function:

```
#define  USB_DEVICE_IDLE                    (0)
#define  USB_DEVICE_ATTACHED                (1)
#define  USB_DEVICE_CONFIGURED              (2)
#define  USB_DEVICE_SET_INTERFACE_STARTED   (3)
#define  USB_DEVICE_INTERFACE_OPENED        (4)
#define  USB_DEVICE_SETTING_PROTOCOL        (5)
#define  USB_DEVICE_INUSE                   (6)
#define  USB_DEVICE_DETACHED                (7)
#define  USB_DEVICE_OTHER                   (8)
#define  USB_DEVICE_INTERFACE_CLOSED        (9)

/* for state change */
#define  USB_STATE_CHANGE_ATTACHED          (0x01)
#define  USB_STATE_CHANGE_OPENED            (0x02)
#define  USB_STATE_CHANGE_DETACHED          (0x04)
#define  USB_STATE_CHANGE_IDLE              (0x08)
```

8.3 Also in ***os_task.c***¸ declare next global variables:

```
/*****************************************************************************
 Global variables
 *****************************************************************************/
usb_device_interface_struct_t*
g_interface_info[USBCFG_MAX_INSTANCE][USBCFG_HOST_MAX_INTERFACE_PER_CONFIGURATION];
uint8_t g_interface_number[USBCFG_MAX_INSTANCE] = { 0 };
```

8.4 Add the prototype for mass_get_interface function that will be used for usb_host_mass_device_event. It is defined in os_task.c just after previous global variable:

```
usb_interface_descriptor_handle mass_get_interface(uint8_t num);
```

8.5 It is time to define the usb_host_mass_device_event function that will handle USB events for the application. This usb_host_mass_device_event requires a function called mass_get_interface. Both functions are defined in ***os_task.c*** as follows:

```
/*FUNCTION*----------------------------------------------------------------
 *
 * Function Name  : usb_host_mass_device_event
 * Returned Value : usb_status
 * Comments       : Called when mass storage device has been attached, detached, etc.
 *
 *END*----------------------------------------------------------------------*/
usb_status usb_host_mass_device_event(
/* [IN] pointer to device instance */
usb_device_instance_handle dev_handle,

/* [IN] pointer to interface descriptor */
usb_interface_descriptor_handle intf_handle,

/* [IN] code number for event causing callback */
uint32_t event_code) { /* Body */
    usb_device_interface_struct_t* pHostIntf =
            (usb_device_interface_struct_t*) intf_handle;
    interface_descriptor_t* intf_ptr = pHostIntf->lpinterfaceDesc;
    volatile device_struct_t* mass_device_ptr = NULL;
    uint8_t i = 0;

    for (i = 0; i < USBCFG_MAX_INSTANCE; i++) {
        if (g_mass_device[i].dev_handle == dev_handle) {
            mass_device_ptr = &g_mass_device[i];
            break;
        }
    }

    if (NULL == mass_device_ptr) {
        for (i = 0; i < USBCFG_MAX_INSTANCE; i++) {
            if (USB_DEVICE_IDLE == g_mass_device[i].dev_state) {
                mass_device_ptr = &g_mass_device[i];
                break;
            }
        }
    }

    if (NULL == mass_device_ptr) {
        USB_PRINTF("Access devices is full.\r\n");
        return USBERR_BAD_STATUS;
    }

    switch (event_code) {
    case USB_ATTACH_EVENT:
        g_interface_info[i][g_interface_number[i]] = pHostIntf;
        g_interface_number[i]++;
        USB_PRINTF("----- Attach Event -----\r\n");
```

```
        USB_PRINTF("State = %d", mass_device_ptr->dev_state);
        USB_PRINTF("  Interface Number = %d", intf_ptr->bInterfaceNumber);
        USB_PRINTF("  Alternate Setting = %d", intf_ptr->bAlternateSetting);
        USB_PRINTF("  Class = %d", intf_ptr->bInterfaceClass);
        USB_PRINTF("  SubClass = %d", intf_ptr->bInterfaceSubClass);
        USB_PRINTF("  Protocol = %d\r\n", intf_ptr->bInterfaceProtocol);
        break;
        /* Drop through into attach, same processing */

    case USB_CONFIG_EVENT:
        if (mass_device_ptr->dev_state == USB_DEVICE_IDLE) {
            mass_device_ptr->dev_handle = dev_handle;
            mass_device_ptr->intf_handle = mass_get_interface(i);
            mass_device_ptr->state_change |= USB_STATE_CHANGE_ATTACHED;
        } else {
            USB_PRINTF(
                    "Mass Storage Device is already attached - DEV_STATE = %d\r\n",
                    mass_device_ptr->dev_state);
        } /* EndIf */
        break;

    case USB_INTF_OPENED_EVENT:
        USB_PRINTF("----- Interface opened Event -----\r\n");
        mass_device_ptr->state_change |= USB_STATE_CHANGE_OPENED;
        break;

    case USB_DETACH_EVENT:
        /* Use only the interface with desired protocol */
        USB_PRINTF("----- Detach Event -----\r\n");
        USB_PRINTF("State = %d", mass_device_ptr->dev_state);
        USB_PRINTF("  Interface Number = %d", intf_ptr->bInterfaceNumber);
        USB_PRINTF("  Alternate Setting = %d", intf_ptr->bAlternateSetting);
        USB_PRINTF("  Class = %d", intf_ptr->bInterfaceClass);
        USB_PRINTF("  SubClass = %d", intf_ptr->bInterfaceSubClass);
        USB_PRINTF("  Protocol = %d\r\n", intf_ptr->bInterfaceProtocol);
        g_interface_number[i] = 0;
        mass_device_ptr->state_change |= USB_STATE_CHANGE_DETACHED;
        break;
    default:
        USB_PRINTF("Mass Storage Device state = %d??\r\n",
                mass_device_ptr->dev_state);
        mass_device_ptr->state_change |= USB_STATE_CHANGE_IDLE;
        break;
    } /* EndSwitch */

    return USB_OK;
} /* Endbody */

usb_interface_descriptor_handle mass_get_interface(uint8_t num) {
    return (usb_interface_descriptor_handle) (g_interface_info[num][0]);
}
```

8.6 Go to ***Events.c*** file and call this usb_host_mass_device_event function inside usbFmw1_ClassDriver0 callback:

```
usb_status usbFmw1_ClassDriver0(usb_device_instance_handle dev_handle,
usb_interface_descriptor_handle intf_handle, uint32_t event_code)
{
    /* Write your code here ... */
    return usb_host_mass_device_event(dev_handle, intf_handle, event_code);
}
```

8.7 In this same ***Events.c*** file, include reference for USB Host HUB functions by adding `#include "usb_host_hub_sm.h"` and edit usbFmw1_ClassDriver2 as shown next:

```
usb_status usbFmw1_ClassDriver2(usb_device_instance_handle dev_handle,
usb_interface_descriptor_handle intf_handle, uint32_t event_code)
{
  /* Write your code here ... */
  return usb_host_hub_device_event(dev_handle, intf_handle, event_code);
}
```

At this point, events for Mass Storage Devices (and HUB) will be handled correctly, however, there is still left to define the application task that will be in charge of mounting the file system once the USB thumb is attached. For this purpose, *APP_USB_task* will be used.

8.8 Go to ***os_tasks.c*** and add prototype for function that will be used to update the state of the USB thumb. Add this prototype just after **mass_get_interface** is declared:

```
usb_interface_descriptor_handle mass_get_interface(uint8_t num);
static void update_state(void);
```

8.9 Now, define this function in ***os_task.c*** file as follows:

```
static void update_state(void)
{
    for (uint8_t i = 0; i < USBCFG_MAX_INSTANCE; i++)
    {
        if (g_mass_device[i].state_change != 0)
        {
            if (g_mass_device[i].state_change & USB_STATE_CHANGE_ATTACHED)
            {
                if (g_mass_device[i].dev_state == USB_DEVICE_IDLE)
                {
                    g_mass_device[i].dev_state = USB_DEVICE_ATTACHED;
                }
                g_mass_device[i].state_change &= ~(USB_STATE_CHANGE_ATTACHED);
            }
            if (g_mass_device[i].state_change & USB_STATE_CHANGE_OPENED)
            {
                if (g_mass_device[i].dev_state != USB_DEVICE_DETACHED)
                {
                    g_mass_device[i].dev_state = USB_DEVICE_INTERFACE_OPENED;
                }
                g_mass_device[i].state_change &= ~(USB_STATE_CHANGE_OPENED);
            }
            if (g_mass_device[i].state_change & USB_STATE_CHANGE_DETACHED)
            {
                g_mass_device[i].dev_state = USB_DEVICE_DETACHED;
                g_mass_device[i].state_change &= ~(USB_STATE_CHANGE_DETACHED);
            }
            if (g_mass_device[i].state_change & USB_STATE_CHANGE_IDLE)
            {
                g_mass_device[i].dev_state = USB_DEVICE_IDLE;
                g_mass_device[i].state_change &= ~(USB_STATE_CHANGE_IDLE);
            }
        }
    }
}
```

8.10 Finally, modify *APP_USB_task's* content as shown next*:*

```
void APP_USB_task(os_task_param_t task_init_data)
{
    /* Write your local variable definition here */
    usb_status status = USB_OK;
    static uint8_t fat_task_flag[USBCFG_MAX_INSTANCE] = { 0 };
    uint8_t i = 0;
#ifdef PEX_USE_RTOS
  while (1) {
#endif
    /* Write your code here ... */
    /* update state for not app_task context */
    update_state();
    /*----------------------------------------------------**
     ** Infinite loop, waiting for events requiring action **
     **----------------------------------------------------*/
```

```
    for (i = 0; i < USBCFG_MAX_INSTANCE; i++)
    {
        switch(g_mass_device[i].dev_state)
        {
        case USB_DEVICE_IDLE:
            break;
        case USB_DEVICE_ATTACHED:
            USB_PRINTF("Mass Storage Device Attached\r\n");
            g_mass_device[i].dev_state = USB_DEVICE_SET_INTERFACE_STARTED;
            status = usb_host_open_dev_interface(usbFmw1_Handler, g_mass_device[i].dev_handle,
g_mass_device[i].intf_handle, (usb_class_handle*) &g_mass_device[i].class_handle);
            if (status != USB_OK)
            {
                USB_PRINTF("\r\nError in _usb_hostdev_open_interface: %x\r\n", status);
                return;
            } /* Endif */
            /* Can run fat task */
            fat_task_flag[i] = 1;
            break;
        case USB_DEVICE_SET_INTERFACE_STARTED:
            break;
        case USB_DEVICE_INTERFACE_OPENED:
            if (1 == fat_task_flag[i])
            {
                g_mass_device_new_index = i;
                /* USB disk is connected and opened, install FS */
            }
            /* Disable flag to run FAT task */
            fat_task_flag[i] = 0;
            break;
        case USB_DEVICE_DETACHED:
            USB_PRINTF("\r\nMass Storage Device Detached\r\n");

            status = usb_host_close_dev_interface(usbFmw1_Handler, g_mass_device[i].dev_handle,
g_mass_device[i].intf_handle, g_mass_device[i].class_handle);
            if (status != USB_OK)
            {
                USB_PRINTF("error in _usb_hostdev_close_interface %x\r\n", status);
            }
            g_mass_device[i].intf_handle = NULL;
            g_mass_device[i].class_handle = NULL;
            USB_PRINTF("Going to idle state\r\n");
            g_mass_device[i].dev_state = USB_DEVICE_IDLE;
            break;
        case USB_DEVICE_OTHER:
            break;
        default:
            USB_PRINTF("Unknown Mass Storage Device State = %d\r\n",
                g_mass_device[i].dev_state);
            break;
        } /* Endswitch */
    }
#ifdef PEX_USE_RTOS
  }
#endif
}
```

8.11 Now, your application is ready to install a File System once the USB thumb is attached. For doing a quick test, msd_fat_demo.c will be added to this project in order to test basic fatfs commands. Copy this file into *Sources* folder. This file is located at: <KSDK_1_3_PATH>\examples\frdmk64f\demo_apps\usb\host\msd\msd_fatfs

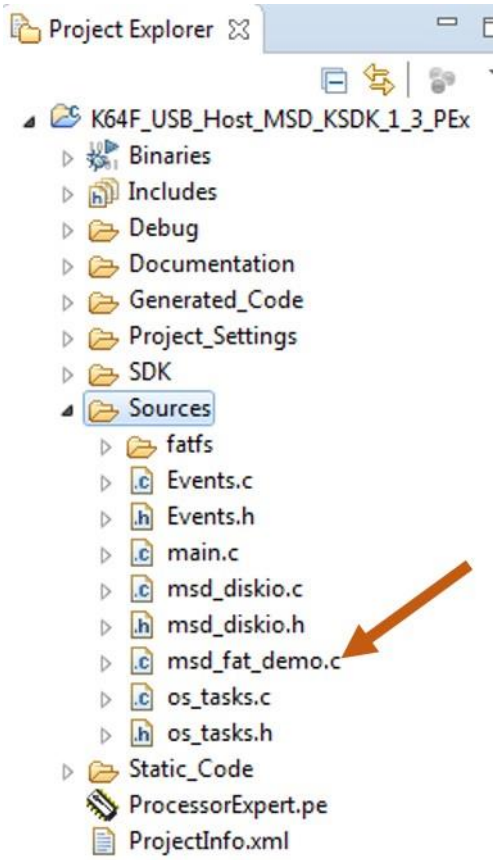

Figure 26 - Add msd_fat_demo.c file

8.12 Call the fat_demo function in *APP_USB_task* (***os_task.c*** file) once device interface is opened:

```
        case USB_DEVICE_INTERFACE_OPENED:
            if (1 == fat_task_flag[i])
            {
                g_mass_device_new_index = i;
                /* USB disk is connected and opened, install FS */
                fat_demo();
            }
            /* Disable flag to run FAT task */
            fat_task_flag[i] = 0;
            break;
```

Note: Remember to declare fat_demo function as extern (at the top of the ***os_tasks.c***) due this function is defined in msd_fat_demo.c file:

```
extern int fat_demo(void);
```

8.13 Lastly, print a message to inform user that program is waiting for any USB thumb to be attached. Go to main.c file and add next code line:

```
int main(void)
/*lint -restore Enable MISRA rule (6.3) checking. */
{
  /* Write your local variable definition here */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  /* Write your code here */
  /* For example: for(;;) { } */
  USB_PRINTF("Waiting for USB to be attached...\r\n");
  /*** Don't write any code pass this line, or it will be deleted during code
generation. ***/
  /*** RTOS startup code. Macro PEX_RTOS_START is defined by the RTOS component.
DON'T MODIFY THIS CODE!!! ***/
  #ifdef PEX_RTOS_START
    PEX_RTOS_START();                  /* Startup of the selected RTOS. Macro is
defined by the RTOS component. */
  #endif
  /*** End of RTOS startup code.  ***/
  /*** Processor Expert end of main routine. DON'T MODIFY THIS CODE!!! ***/
  for(;;){}
} /*** End of main routine. DO NOT MODIFY THIS TEXT!!! ***/
```

8.14 Save changes and compile the project. Everything should be ok.

9 Memory Allocation Considerations

If you download the project as it is, application will not detect USB events. It is because USB Host stack is not initialized correctly due dynamic memory allocation failures. By default, Processor Expert add the ***fsl_misc_utilities.c*** file and in this file, the **_sbrk** function is defined. This function is used to allocate memory, however, by default there is not heap size for the project, so every memory allocation fails. There are two possible solutions to this situation:

- ***Exclude fsl_misc_utilities.c from build configuration***.
  If fsl_misc_utilities.c file is excluded from current build configuration, memory allocation utilities for GCC toolchain are used. In these utilities, there is a heap size defined by default so memory allocation problems disappear.
  For this solution, just locate the ***fsl_misc_utilities.c*** file (***SDK/platform/utilities/src***) and right click on this file, then select *Resource configuration* and *Exclude from build…*. After this, a new window will appear with those configurations for which this file will be excluded. Select the debug configuration and press the Ok button.

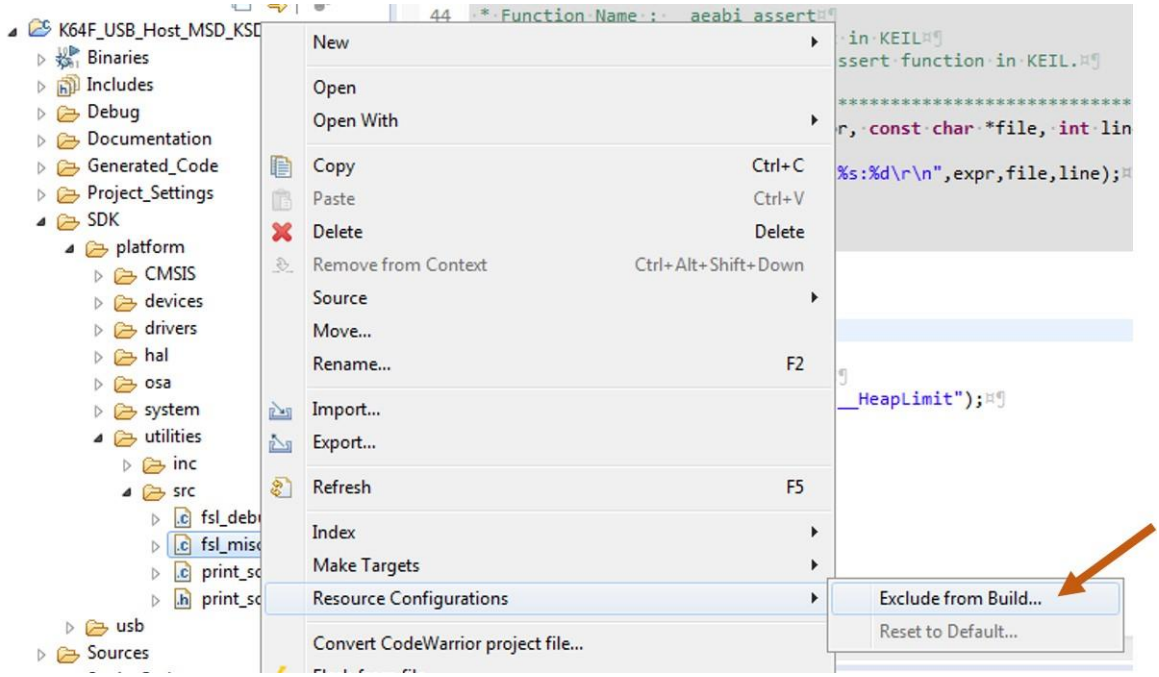

Figure 27 - Exclude fsl_misc_utilities.c from build

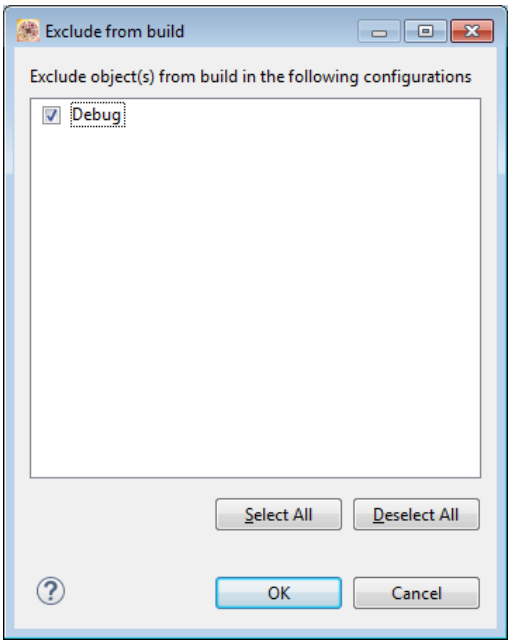

Figure 28 - Exclude from Debug configuration

Recompile the project and it should be working well.

- ***Add heap size to linker flags in order to use fsl_misc_utilities.c file.***
  The second solution is to add a HEAP size definition in linker configuration so fsl_misc_utilities should not be excluded, just right click under project and select "properties", then, on C/C++ Build, select the Settings option and then locate Cross ARM C++ Linker. In this section, you can see the Miscellaneous option and a field where you can add next definiton ***-Xlinker --defsym=__heap_size__=0x1000***. Add this HEAP definition and then compile the project again and it should work without excluding fsl_misc_utilities.c file.

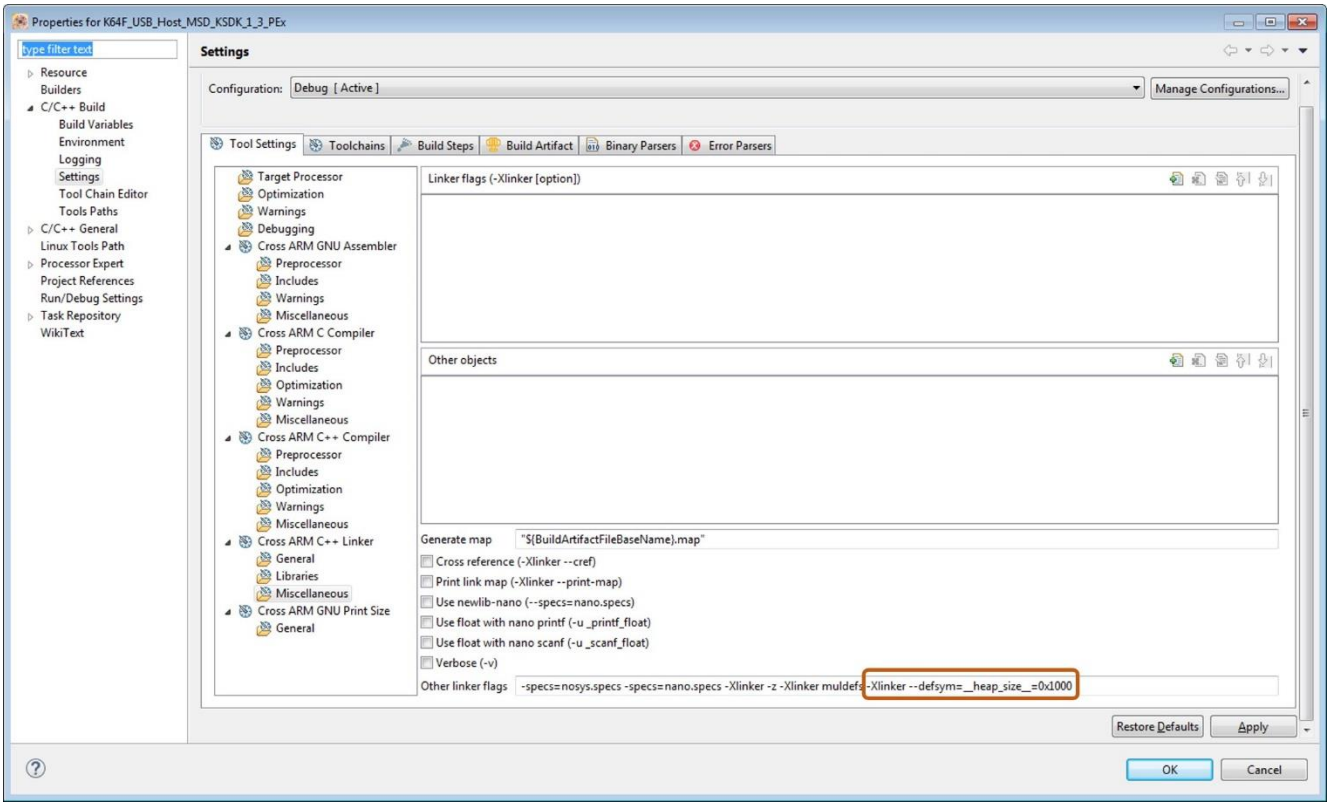

Figure 29 - Define HEAP size for project

Recompile the project and it should be working well.

Once you select one of these two options, the project should work well.

10 Run the Project

Once memory allocation problems are removed, it is time to test the project. Compile the project and download it into FRDM-K64F board, then, do next configurations:

10.1 For USB Host functionality in FRDM-K64F, Jumper J21 should be connected. Remember that this connection is not protected electrically, so, be careful when connecting the USB device in order to avoid damage on the board:

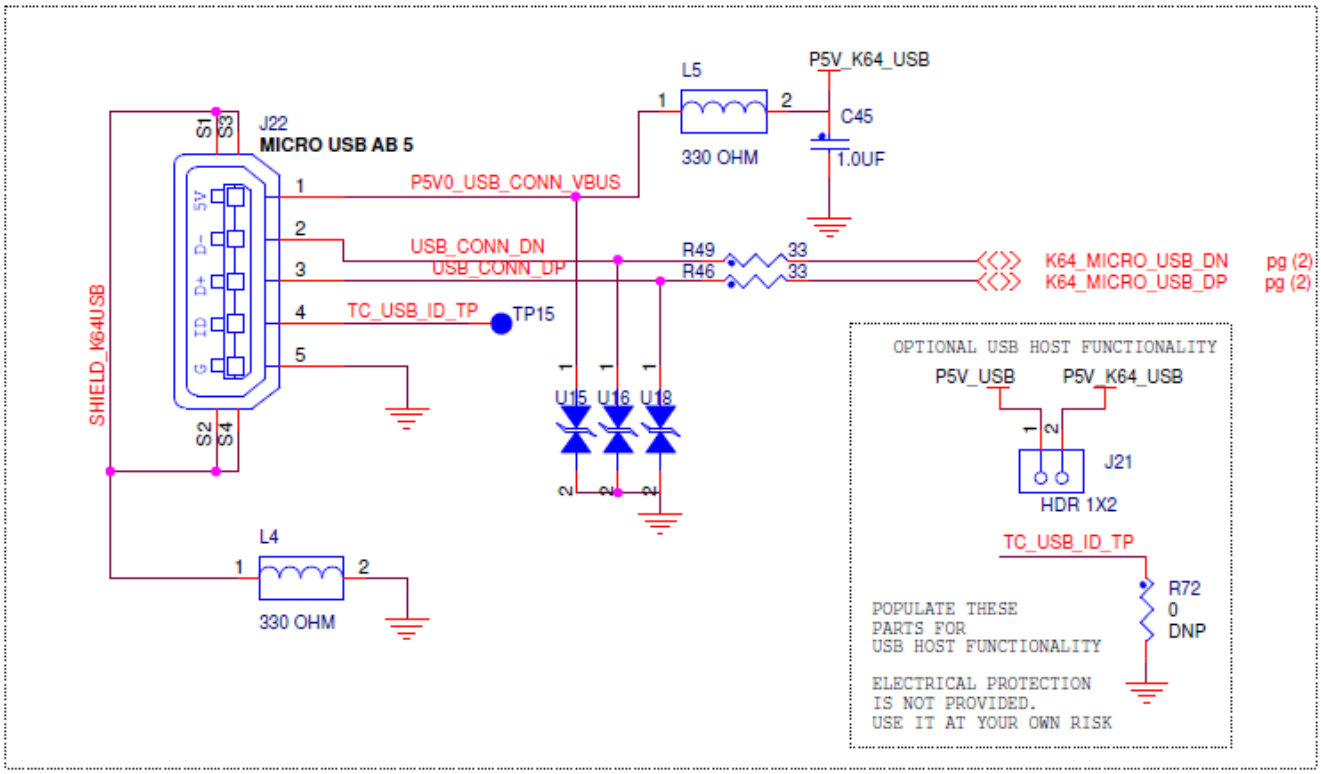

Figure 30 - USB connection on FRDM-K64F

10.2 Connect to USB Connector (J22) a Micro USB OTG adaptor just as the one shown below:

Figure 31 - Micro USB OTG adaptor

10.3 Open a terminal to 115200 bps.

10.4 Insert an USB thumb on Micro USB OTG adaptor and see the message that is printed onto the terminal, File System test should be passed correctly.

Figure 32 - Project working correctly

11 Conclusion

A basic application for USB Host with MSC, fatfs and Processor Expert support has been created in order to serve as guidance for any other USB class or MSD specific implementation. This project is using Non-RTOS (bare-metal) implementation, however, it is capable to add RTOS support just like MQX or FreeRTOS by modifying OS field in fsl_os_abstraction component and adjusting some other settings specific for every RTOS (like disabling debug_console component in MQX).

For more USB Host example's reference please check those located at: <KSDK_1.3.0_PATH>\examples\<BOARD>\demo_apps\usb\host.

If more information related to USB Host APIs are needed, refer to *USB Stack Host Reference Manual.pdf* document that is included at <KSDK_1_3_PATH>\doc\usb.