Q. What are the different methods to integrate additional files with existing component?

A. Method to be used for integrating the additional files depends on the component requirements. Broadly they can be classified into the following categories:

1. **File processing architecture**

   Under this hood, user can select the option based on the architecture which their library files follow.

   i. Process the information – This option is more or less same as including the standard library files like stdlib.h or stdio.h etc. in a project. The additional files used in this case are the software files which process the information and provide the data back. Typical example for such applications is encryption or decryption of the data.

   With this approach user have to make sure that no function in the library should make a call to any of the functions which depend on the controller resources for its operation. All the functions inside the library file must be self sufficient.

   ii. Process the information and control the PHY layer as well – This option is used when additional files available does need the controller resources for them to process the information. Typical example being WiFi module where it needs to send some information to configure/control with the host.

   Problem with this approach is the hardware resources will have different names based on the instance name of the component. As user may change the name of the component according to his naming requirements but the library files will make a call to a fixed function. Thus, to resolve this conflict we need a mapping layer which will map the calls from library files to the hardware functions. Below provided example shows that the additional files make a call to function SPIM_WriteData and in turn this function makes a call to the function with an appropriate name.

```
void SPIM_WriteData(uint8 txData)
{
        `$INSTANCE_NAME`_SPIM_WriteTxData(txData);
}
```

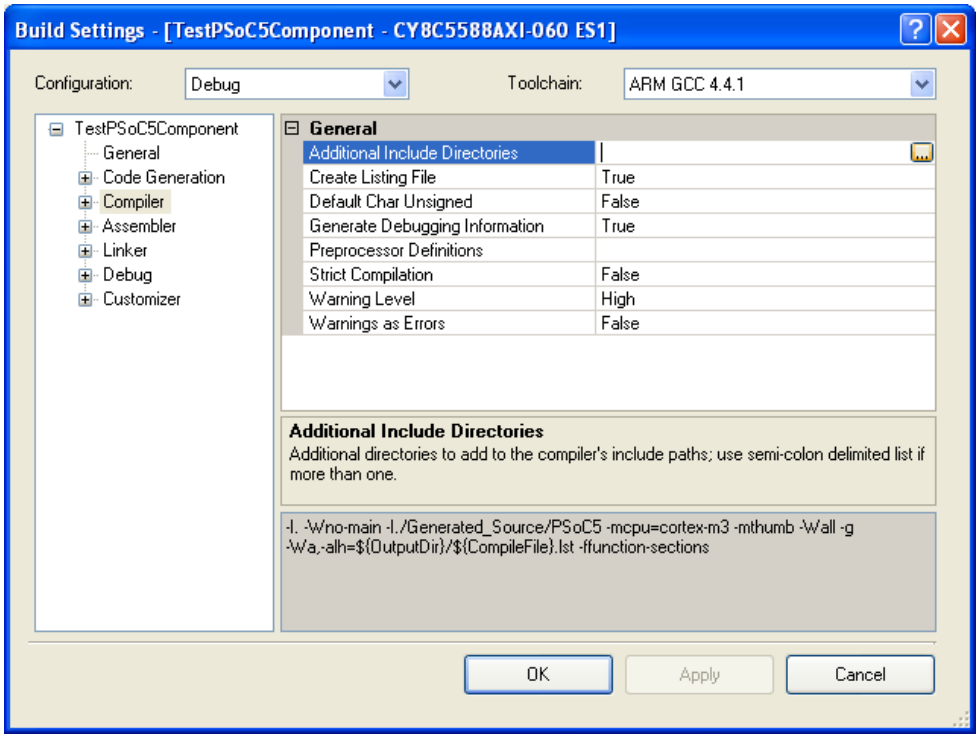
   This mapping file is made as a part of the component, thus `$INSTANCE_NAME` will be replaced with the name of the component.

2. **Visibility of source code to user**

   Many a times, user does not want to share the source code of the additional files either to protect their code or they don't want user to modify their code which makes debugging in

case of a problem a lot more difficult. On this basis, integration of additional library files again can be of two different types:

i. Library file/Include Directory – When this approach is followed, all the source files are visible to the user. This is the simplest of all and customer can make calls to the functions available in the library files from both Main project and component as well without any additional headache. All the user has to do is include the additional directories in his project. This can be done from Project → Build Settings → Compiler. Below snapshot shows the snapshot for the same.



This option is exactly similar to the standard project architecture, where function prototypes are declared in some file and accessed from another one. In this mode, all the PHY functions are available to these files automatically.

ii. Object Library – This option is used by most of the users as it hides the source code from the end user. When this option is used, all the library files are compiled separately and they should not include any header file which belongs to the project. If there are any controller related functions which had to be called from these files, then a prototype of the same has to be declared in these additional files and project must be compiled. During the linking process these object files are linked with the main project and the function prototypes are replaced with their definitions.

Q. How to declare variables in library files which are used with the component?

A. If there are any variables which are available in library files and have to be used by the component, then in the component files those variables are declared as extern. For e.g. in the statement below dbgPrint is the variable present inside the library files but are used in the component.

```
extern uint16     dbgPrint;
```

By declaring this variable as extern, compiler is instructed that the definition will be provided during the run time. Thus, linker resolves the definition of this variable.

If there are variables of certain typedefs in library files, then there those same typedefs have to be declared again in the component and then same extern variable has to be used.

Q. How to access the APIs of the components used in designing the custom component from the library files?

A. In order to access the APIs of the components from the library files then with the use of mapper file any component functions can be accessed. This mapper file routes the function call from a fixed name to the instance specific function. Usage of this file, provides the freedome to the end user of having his own name for the component.

This mapper file is present inside the component and contains the functions which are called from library files. For example, any function in library can call the SPIM_WriteData function. This function will in turn make an appropriate call to the component related function.

```
void SPIM_WriteData(uint8 txData)
{
      `$INSTANCE_NAME`_SPIM_WriteTxData(txData);
}
```

In the above example `$INSTACE_NAME` will be replaced with the name of the component.