# Interprocessor Communication (IPC)

# Shared Memory Transport

## Purpose

The goal of this lab is to become familiar with how to use the Interprocessor Communication Module (IPC) to communicate between applications running on different cores. We will build a project that will use the MessageQ module of IPC to pass messages between arbitrary cores on the 6678. Initially, we will be using shared memory in order to pass data between the cores.

Several instructions that are part of the usage of IPC were eliminated from the source of the original code that is used in this Lab. The student will have to add these instructions. Hints what to add are given in the source code next to the work TODO. In addition, students will have access to the correct code as well.

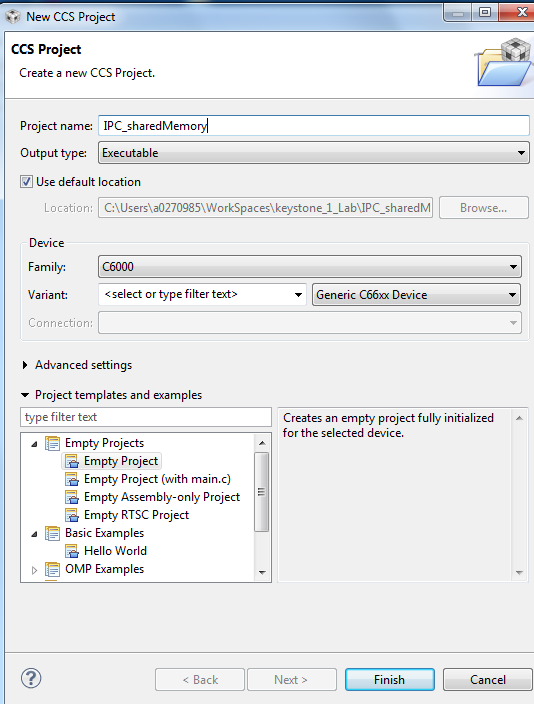
## Project Details

The project will generate a single .out file that will run on all cores. Core 0 is designated the “Master” core, meaning that it will be the one that is responsible for initialization tasks. A token message will be passed between randomly chosen cores 100 times. The current count of the number of passes is part of the token message. When a core receives the token message, it will send an acknowledge back to which ever core the token came from. The core that receives the 100th token message pass will also be responsible for freeing the memory used by the token message and then sending a “Done” message to all of the cores. Upon receipt of the “Done” message, each core will do its cleanup and then exit.

The application is implemented with a single task, and a polling implementation is used. Once initialized, each task will poll it’s MessageQ to see if there is a message waiting. If there is, the message is read, and then action is taken based on the message type.

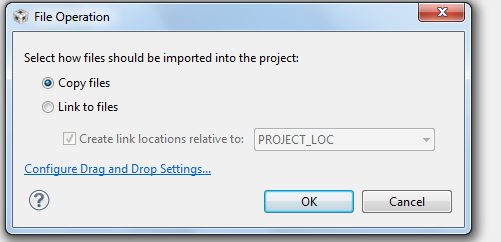
### Task 1: Start CCS new project

1. Start CCS. From the file tab select new->CCS Project and chose empty project with the name IPC\_sharedMemory :

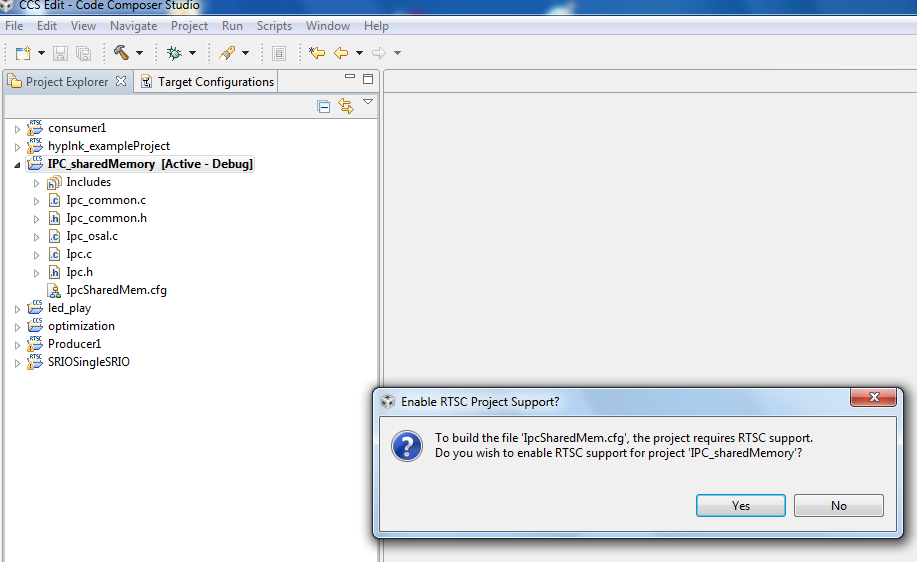


Click finish.

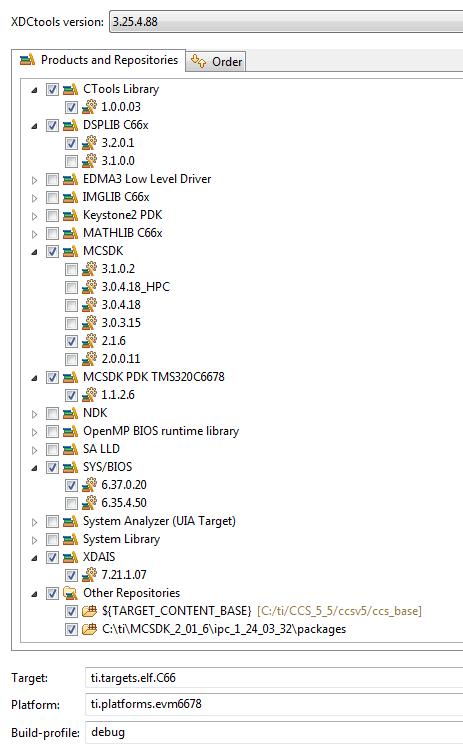
1. Select the new project, right click and choose add files. In the dialogue box navigate to the SharedmemoryOriginal project location (the instructor will give the project location) and select all the files. Ipc.c, Ipc.h, Ipc\_common.c, Ipc\_common.h, Ipc\_osal.c, IpcSharedmem.cfg. Click open. A dialogue box will ask you to copy or to link the files. Select copy and click OK.



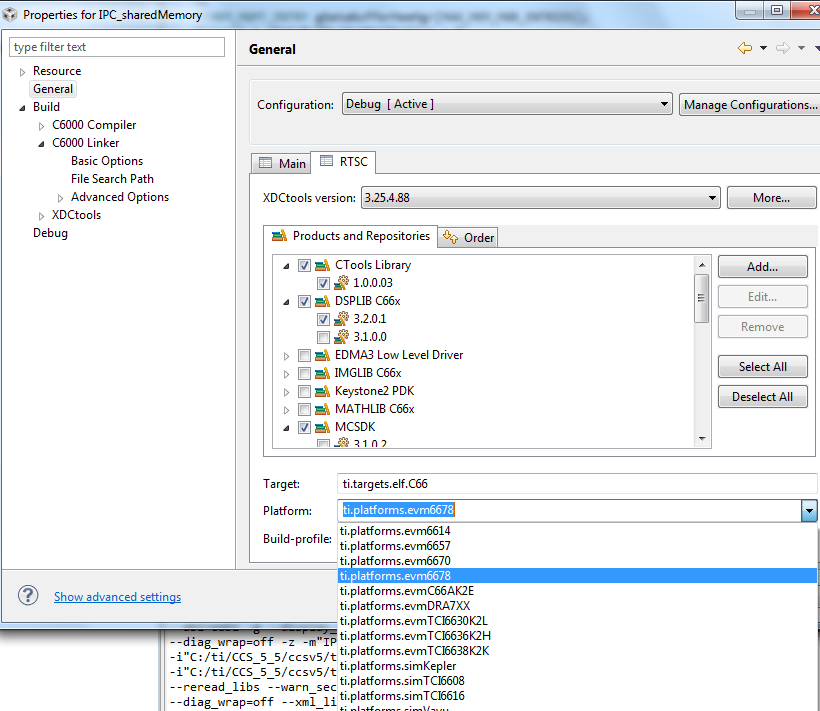
1. As soon as the files were copied into the project, a dialogue box will ask you if you want to start RTSC project. Choose yes and continue



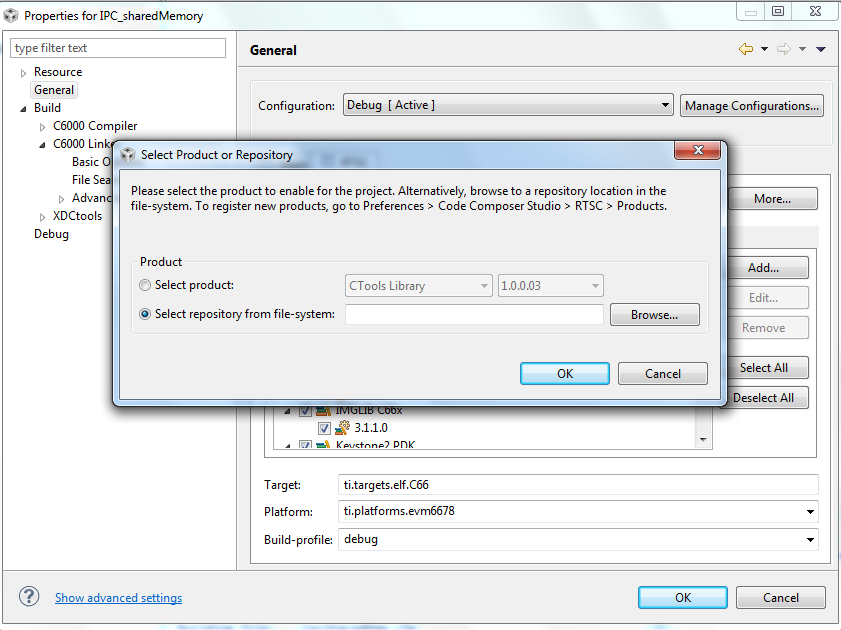
1. Next select the project, right click and select properties.
   1. In the RTC tab (general->RTSC) make sure that ONLY the repositories that are checked in the next screen shot are checked in your project



* 1. If the platform is not set, set the platform by starting the pull down menu (right click on the arrow next to the platform tab. Then choose ti.platforms.evm6678 by clicking twice on the correct platform



* 1. In the RTSC tab, add the ipc repository. From the RTSC window select click on ADD, and choose select repository from file system



Browse to the location of the IPC packages. On my system it is in MCSDK\_2\_01\_6\ipc\_1\_24\_03\_32\packages. If your Laptop has a different release of MCSDK, the path will be changed accordingly

* 1. Click OK
  2. Click OK on the properties. Right click on the project and choose rebuild. Make sure to see that the project is built.

### Task 2: Examine the Skeleton Project

1. Examine the IPC\_sharedMemory.cfg project.
   1. Right Click on the file in the Project Explorer
   2. Choose Open With -> XDCScript Editor
2. Note the following lines in the .cfg that are necessary for the IPC example.
   1. The following lines include the modules that are necessary for using IPC in this manner. The MultiProc module handles the management of the various processor IDs. The IPC module is used to initialize the subsystems of IPC. The HeapBufMP module manages allocation of memory buffers from the shared heap. And the MessageQ module supports the sending and receiving of variable length messages.

**var** MessageQ = xdc.useModule('ti.sdo.ipc.MessageQ');

**var** Ipc = xdc.useModule('ti.sdo.ipc.Ipc');

**var** HeapBufMP = xdc.useModule('ti.sdo.ipc.heaps.HeapBufMP');

**var** MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');

* 1. The following line defines which processors will be used. In this case, we will be using all of them.

Ipc.procSync = Ipc.ProcSync\_ALL;

* 1. These lines define the shared memory location.

**var** SHAREDMEM = 0x0C000000;

**var** SHAREDMEMSIZE = 0x00200000;

* 1. These include the SharedRegion module, which manages the shared memory allocation across processors and defines the specific location of the shared memory.

**var** SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');

SharedRegion.setEntryMeta(0,

{ base: SHAREDMEM,

len: SHAREDMEMSIZE,

ownerProcId: 0,

isValid: **true**,

name: "DDR2 RAM",

})

1. Open and examine Ipc.h. Note specifically the enumeration for the different message types, and the myMsg structure. The exact myMsg structure definition is arbitrary, except for the requirement that a MessageQ\_MsgHeader element is the first item. In this case, the message consists only of a Message header, a message type, and an int that holds the count of the number of times that the token has been passed.
2. Open and Examine Ipc.c. There are 3 functions.
   1. Main – Dynamically creates the application task, calls Ipc\_start to synchronize the processors, and then calls Bios\_start. Nothing magical here.
   2. findNextCore – This functions is very simple. It just generates a random number between 0 and MAX\_NUM\_CORES -1. It ensures that the generated number is different than the current core so that none of the cores are passing the token to themselves*. Note: There is no reason that a core can’t send a message to itself in this manner. In fact we do that at the end with the “Done” message. But we prevent the token passing from doing this so that the output shows a more interesting token path.*
   3. task\_fxn - This is where all of the magic happens. The Master Core does the initialization, and creates the shared heap, and all of the slave cores connect to it. All cores register the heap with MessageQ, and they create a “Local” MessageQ where their messages will be received. Each core creates a small lookup table that associates the core number with the appropriate MessageQ Id. Finally, Core 0 creates the token message, passes it to a random core, and then all cores just wait for messages to be received and acted on.

### Task 3: Add IPC API’s

1. Some of the APIs have been left out of task\_fxn. Each location where source needs to be added is marked with the comment “TODO: IPC #<x> -” followed by a description of the task. The <x> holds the task number. Each element of the code that needs to be added is a single function call.

TIP: CCSv5 has a Tasks window that will give shortcuts to these tasks. One way to get to it is from the Window->Show View->Other menu in CCS. When that window opens, look under the “General” folder and double click “Tasks”. You can then double click on any of the tasks and you will immediately be taken to the source where that task is located.

#### Hints:

The following are hints about the code to be added. Try to add the code using only the descriptions contained within the source code. If you need more clues, use the hints below.

* 1. Allocate Memory for the Token Message
     + The memory for a MessageQ message is allocated by calling the API MessageQ\_alloc(). The parameters are the ID of the heap that the memory will be allocated from, and the size of the message to be allocated.
  2. Pass the token to the destination core.
     + MessageQ messages are passed by calling the MessageQ\_put() API. The parameters to the function are the destination Queue Id, and the pointer to the message.
  3. Get a Message from the local queue.
     + Messages are retrieved from the MessageQ with the MessageQ\_get() API. The MessageQ\_get() function is a blocking call. The parameters it accepts are the handle of the MessageQ to check, the *address* of the pointer to the message, and an enumerated parameter that specifies how long to block until a timeout occurs. MessageQ\_FOREVER specifies that a timeout will never occur. The return value is zero if the message is successfully retrieved.
  4. Get the Reply Queue ID for the token.
     + The reply queue Id is obtained via the MessageQ\_getReplyQueue() API. The only parameter needed is a pointer to the message. The return value is of type MessageQ\_QueueId.
  5. Allocate the acknowledge message
     + See Hint #1
  6. Send the acknowledge message.
     + See Hint #2
  7. Free the memory used by the token message.
     + The memory is freed by the API MessageQ\_free. The only parameter is the pointer to the message.
  8. Note that the loop that sends the “Done” message will send one of those messages to itself.

Set the Reply Queue for the Token Message

* + - This is done using the MessageQ\_setReplyQueue() API. Parameters are the handle to the MessageQ and the pointer to the message.

### Task 4: Build and Run the application

1. Once everything has been added to Ipc.c, build the project. It should build without errors or warnings. If it doesn’t build properly, attempt to figure out why. Otherwise, ask the instructor.
2. Launch the Debug Session and Connect to all of the cores.
3. Load Ipc.out to all of the cores.
4. Run All of the Cores

### Task 4: Verify the Output

1. When the device runs, you should see printf output in the console window that looks like the output below. The order of your output is expected to be different since the token is being passed randomly.

[C66xx\_1] Token Received - Count = 1

[C66xx\_0] Ack Received

[C66xx\_5] Token Received - Count = 2

[C66xx\_1] Ack Received

[C66xx\_2] Token Received - Count = 3

[C66xx\_1] Token Received - Count = 4

[C66xx\_5] Ack Received

[C66xx\_5] Token Received - Count = 5

[C66xx\_1] Ack Received

[C66xx\_2] Ack Received

[C66xx\_3] Token Received - Count = 6

1. Ensure that the application behaves as expected, by checking the following items.
   1. Since Core 0 is passes the token first, the first Ack received message should be by Core 0. This is the case with the example above.
   2. In general, you should see the pattern of how the token was passed. We know that Core 0 passes the first token. From the output above, it looks like The order is 0 -> 1 -> 5 -> 1 -> 2 -> 1 -> 5 -> 3, because that’s the order of the Token receive messages. The Ack Received messages should follow a similar, pattern, but it likely won’t look identical. (This can be attributed to the non-realtime nature of the printf function, but in reality, the pattern would be identical.) In this case, the pattern is identical for the first few passes.
   3. The last Token Message should have a count of 100.
   4. There should be a “Done Received” message from each core in use near the very end.
   5. Each core should halt at the C$$EXIT symbol when the demo is complete.