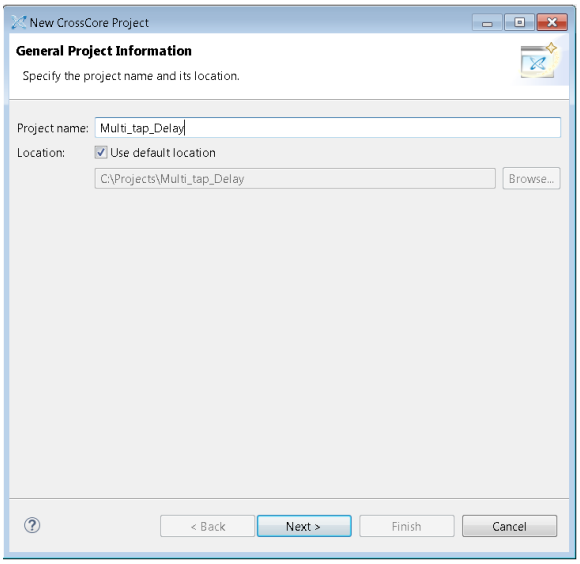
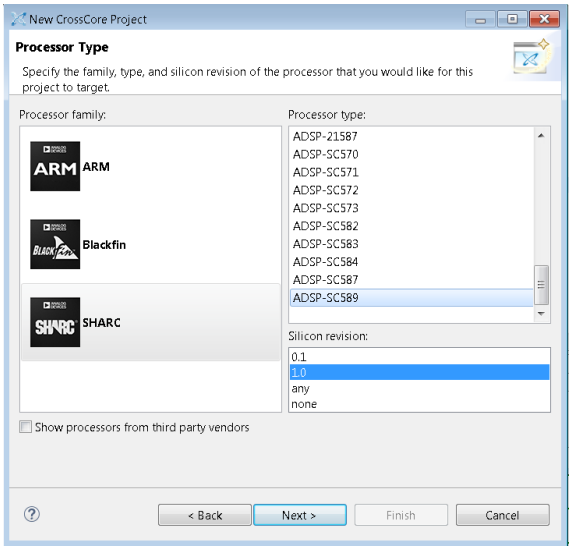
**Step 1: Create a CCES plugin project**

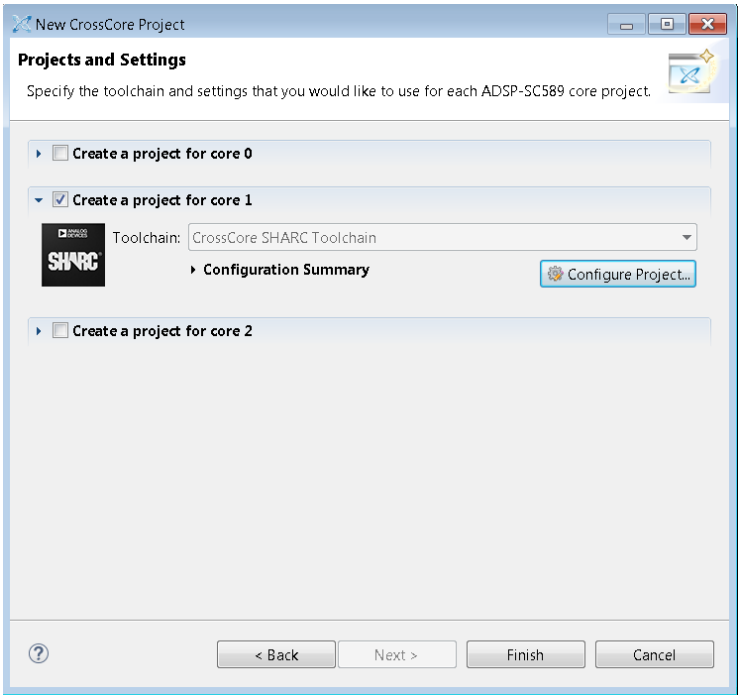
Open CCES. In the menu bar, click *File->New->CrossCore Project*. Input project name, for example, Multi_tab_Delay, in *Project Name*. Users could designate any project directory in *Location*. Click *Next*.



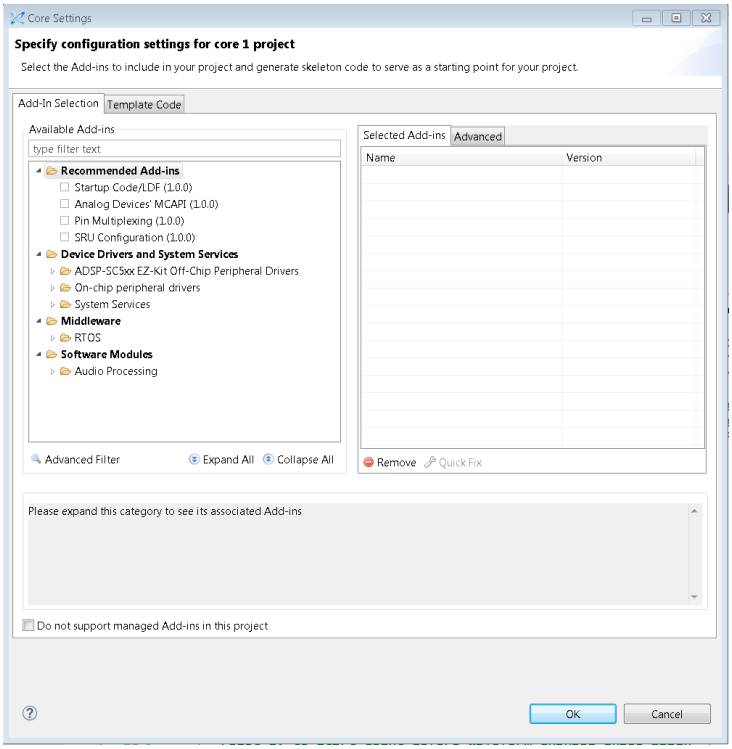

Choose *SHARC* in *Processor family* at the left column, and then select *ADSP-SC589* in *Processor type* at the right column, and select *1.0* in *Silicon revision*. Please note users should choose processor and silicon revision that comply with hardware. Click *Next*.




Uncheck *Create a project for core 0* and *Create a project for core 2*. Because we assume that the whole audio module only runs on a single Sharc core, either Sharc core 1 or Sharc core 2.

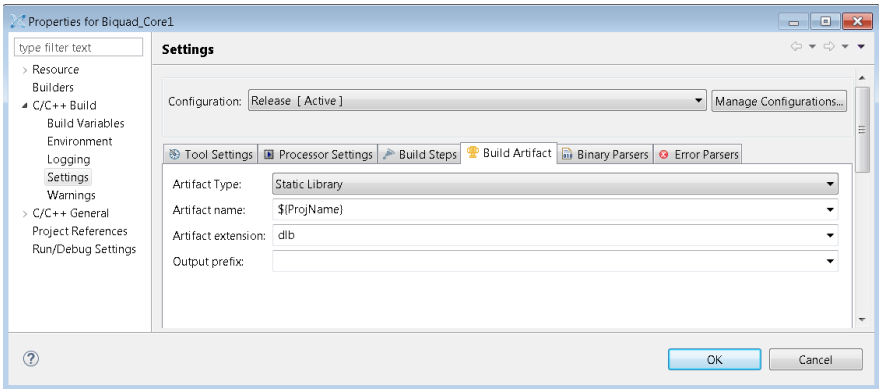



Click *Configure Project*. Uncheck all *Recommended Add-ins*, *Device Drivers and System Services*, *Software Modules*, etc. Click *OK* and come back to the previous dialog window. Click *Finish*. Then you should be able to see a project named "Biquad_Core1" in the CCES *Project Explorer* window.




In the CCES menu bar, click *Project*->*Build Configurations*->*Set Active*->*Release*. The Release Build makes the project use release version libraries and no debug information included.

In the CCES menu bar, click *Project*-> *Properties*. In the left column of the dialogue window, select *C/C++ Build*->*Settings*, then select *Static Library* in the tab *Build Artifact*. Click *OK*.




Copy the file, adi_ss_extmod.h, from "C:\Analog Devices\SoftwareModules\SigmaStudioForSHARC-SH-Rel3.9.0\Host\Include" to the project *src* folder. Then you should see it in the CCES *Project Explorer* window.

Open "Biquad_Core1.c" and add "#include "adi_ss_extmod.h"". The header file includes necessary Sigma Studio related declarations. After that, remove main function, which is not used in the future.

**Step 2: Create wrapper functions and develop algorithms**

SigmaStudio modules can be of two types: Modules supporting sample/stream processing and Modules supporting block processing. This instruction will describe how to create plugins for block processing. The sample based processing uses a similar approach but different wrapper functions and arguments.

The entry-point function must have a prefix 'BPROCESS_' to indicate that it is a block processing Plug-In function for SigmaStudio. Sub-functions in the source file which are referenced only from the main Plug-In function or other sub-functions need not follow any convention. The general prototype of a block processing Plug-In entry-point function is as shown below.

**void BPROCESS_<algorithm_name> (SSBlockAlgo* pBlockAlgo)**

In addition to the process function, block Algorithms can also have an initialization function. Initialization functions are used to initialize the Module in case of decoders and 3rd party post processing Modules. The function name must have the prefix 'INIT_' to indicate that it is the initialization function of the Module. The general prototype of a block processing Plug-In initialization function is as shown below.

**void INIT_<algorithm_name> (SSBlockAlgo* pBlockAlgo)**

Create two functions in "Biquad_Core1.c", void BPROCESS_Biquad (SSBlockAlgo* pBlkAlgoInfo) and void INIT_Biquad (SSBlockAlgo* pBlkAlgoInfo), in compliance with previous requirements.

The struct, SSBlockAlgo, is the interface between Sigma Studio libraries, which is a part of Sigma Studio framework/firmware and runs on Sharc cores, and the custom audio algorithms, as shown below.

```
typedef struct _SSBlockAlgo
```

```
{
    int32_t     nInputs;        /* Number of input pins */
    int32_t     nOutputs;       /* Number of output pins */
    Block      *pInputs;        /* Pointer to array of block i/o mem structure */
    Block      *pOutputs;       /* Pointer to array of block i/o mem structure */

    int32_t     nGrowth;        /* Indicates growth count */
    int32_t     nGrowthB;       /* Secondary growth count */

    void       *pParam;         /* Pointer to parameter memory */
    float32_t  *pState;         /* Pointer to state memory */
    float32_t  *pScratchDM;     /* Pointer to scratch in DM memory */
    float32_t  *pScratchPM;     /* Pointer to scratch in PM memory */
    float32_t  *pStateB;        /* Pointer to state memory B */
    float32_t  *pStateC;        /* Pointer to state memory C */
    float32_t  *pExtPreState;   /* Pointer to extended precision state memory */
    int32_t    *pExtSymbols;    /* Pointer to symbol address table */
    float32_t    *pSharedMem;     /* Pointer to shared memory buffer in L2 */
} SSBlockAlgo;
```

The complete algorithm function is shown below.

```
void BPROCESS_Multi_Tap_Delay (SSBlockAlgo* pBlkAlgoInfo){

      float32_t *params = (float32_t*)(pBlkAlgoInfo->pParam);
      int32_t nDelayLineSize = (int32_t) (params[0]);

      int32_t WritePos = (int32_t)pBlkAlgoInfo->pState[0];

      float32_t *pInput  = pBlkAlgoInfo->pInputs[0].pSamples;
      int32_t nBlockSize  = pBlkAlgoInfo->pInputs[0].pBlockProperties->nBlockSize;

      int32_t RepCount = pBlkAlgoInfo->nGrowth;

      nDelayLineSize += nBlockSize;

      float32_t *DelayLine = pBlkAlgoInfo->pStateC;


      for(int32_t loopcount = 0;loopcount<nBlockSize;loopcount++, WritePos++){
            DelayLine[WritePos%nDelayLineSize] = pInput[loopcount];
      }

      if (WritePos>=nDelayLineSize)
            WritePos-=nDelayLineSize;

      pBlkAlgoInfo->pState[0] = (float32_t) WritePos;

      for(int32_t index = 0; index < RepCount; index++){

            int32_t tap  = (int32_t) pBlkAlgoInfo->pInputs[index+1].pSamples[0];

            if (tap > nDelayLineSize-nBlockSize)
                  tap = nDelayLineSize-nBlockSize;
```

```
                int32_t ReadPos = WritePos - nBlockSize - tap;
                if (ReadPos<0)
                        ReadPos += nDelayLineSize;

                float32_t  *pOutput= pBlkAlgoInfo->pOutputs[index].pSamples;


                for(int32_t loopcount = 0;loopcount<nBlockSize;loopcount++, ReadPos++){
                        pOutput[loopcount] = DelayLine[ReadPos%nDelayLineSize];
                }
        }

}
```

**Step 3: Create XML file**

The Module XML file is an input to the Algorithm Designer and contains the source definitions for the Module which includes name and path of the module binary, parameter details, pin details etc. Each XML can contain Module source definitions for one or more Modules. The following items are included in the source definition.

- Name, path and target processor of DLBs of the Module (ADSP-SC5xx).
- List and details of all the runtime parameters of the Module.
- Details of the input and output pins.

The following is an example of multi_tap_delay module.

```
<?xml version="1.0" standalone="true"?>
- <SS4SH version="3.9.0.0" description="SigmaStudio for SHARC Algorithm Designer" name="ss4sh_module_xml">
  - <module name="Multi_Tap_Delay" block="TRUE">
      <dlb name="Multi_Tap_Delay.dlb" path="..\Project\Release\" embed="FALSE" target="ADSP-SC58x"/>
      <parameter name="DelayCount" format="FLOAT" isbuffer="NO"/>
      <pin direction="INPUT" type="DATA"/>
      <pin direction="INPUT" type="CONTROL"/>
      <pin direction="OUTPUT" type="DATA"/>
    </module>
</SS4SH>
```

The delay module has one parameter for users to control, which is delay line size. Delay tabs are wired into the module as input pins that have type of “control”. Please note the sequence of parameters in the XML determines the sequence in the parameter buffer that is passed to the wrapper function. So does the input/output pins (buffers) sequence in the XML file.

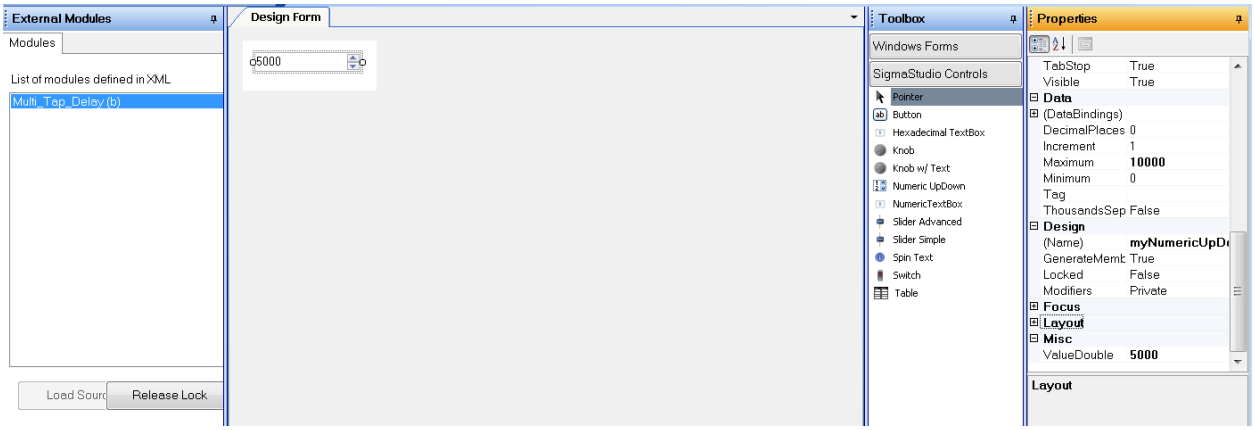
**Step 4: Design graphic user interface**

Algorithm Designer, a component of Sigma Studio, is used in integrating externally developed Algorithms. In the Algorithm Designer, users design controllable interface that connects to the dlb file generated in the previous step. Once the design is finalized, Algorithm Designer generates DLLs (Dynamic Link Libraries) which can be used by Sigma Studio for SHARC for redistributing the Plug-In module.

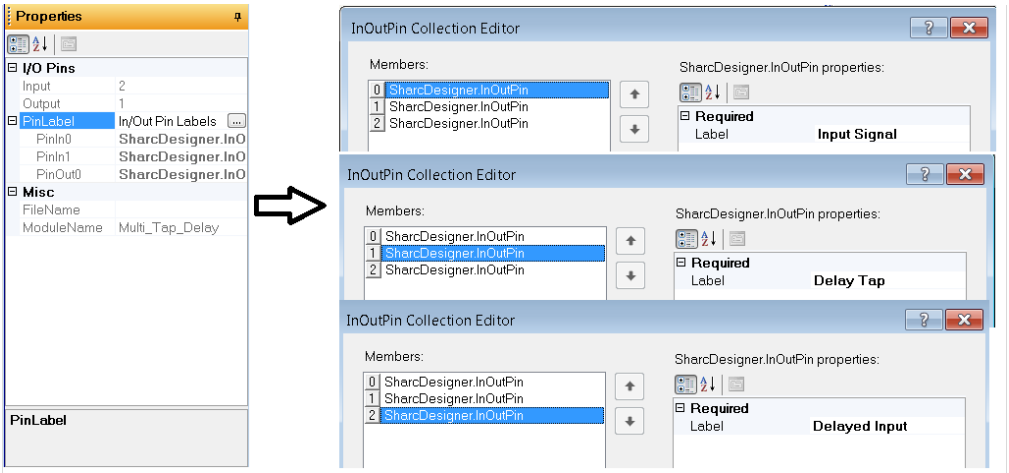
Open Sigma Studio and Click *Action->Launch Algorithm Designer* in the menu bar. In the upper-left window, *External Modules*, click *Load Source* to load in Multi tap delay xml file.

In the upper-middle window, *Design Form*, drag and drop in the widget, *Numeric UpDown*, from the upper right window, *Toolbox*. Then users could modify the widget properties in upper right most window,

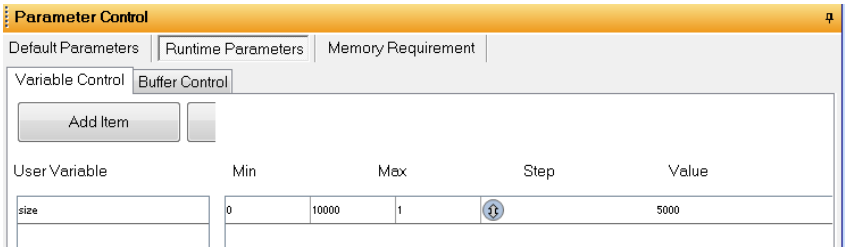
*Properties*. In order to match actual maximum size in processor memory, we set maximum to 10000, minimum to 0 and value to 5000 in the *Properties*.



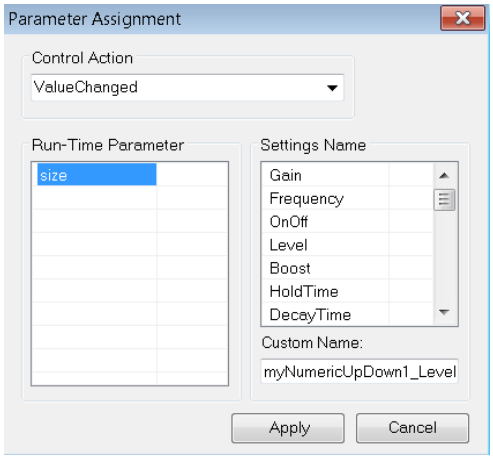

Click Multi_Tap_Delay in *Modules*, then set pin names to “Input Signal”, “Delay Tap”, and “Delayed Input” accordingly. Their sequence is derived from the XML file.



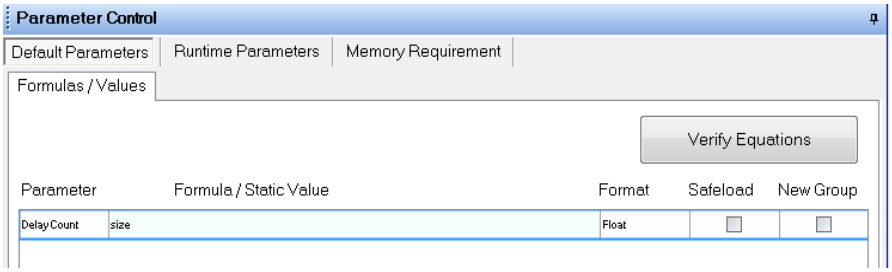

In the lower-left window, *Parameter Control->Runtime Parameters*, add a variable, and rename it “size”, in *Variable Control*. We put 0, 10000, 1 and 5000 in Min, Max, Step and Value respectively.



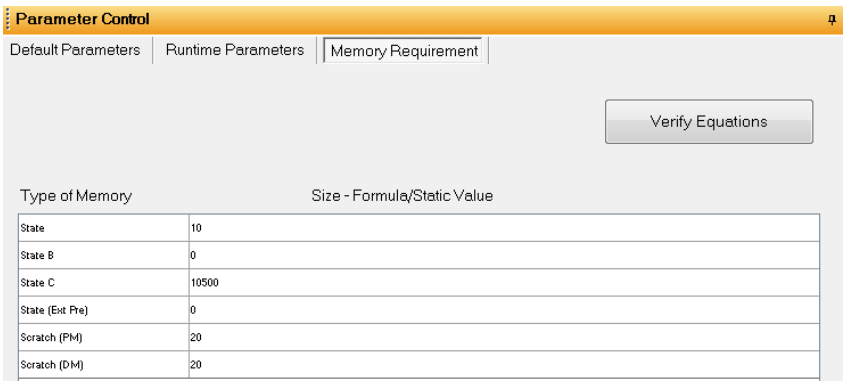

The next step is to connect the “*Numeric UpDown*” widget to the created variable “*size*”. Click mouse right button on the “*Numeric UpDown*” widget in the *Design Form*. Choose *Assign* and you should see a similar window shown below. Set *Control Action* to *CheckedChanged* and *Run-Time Parameter* to the variable, enable. Click *Apply*.




To connect the widget to the parameter "*DelayCount*" in the dlb file, in the lower-left window, *Parameter Control->Default Parameters*, put in the variable "*size*", in Formula / Static Value, as shown below.




The custom module requires State and Scratch memory in execution. Users should calculate memory size based on actual needs. The delay example demands two variables for storing the write and read pointers, which is allocated in State memory and multiple local variables, which is allocated in Scratch memory, as shown below. The allocated memory must be equal or greater than the actual needed memory. The State C memory locates on external memory and will be used for hosting delay line buffer. Its size need to be equal or greater than the maximum value 10000 we put in the widget.

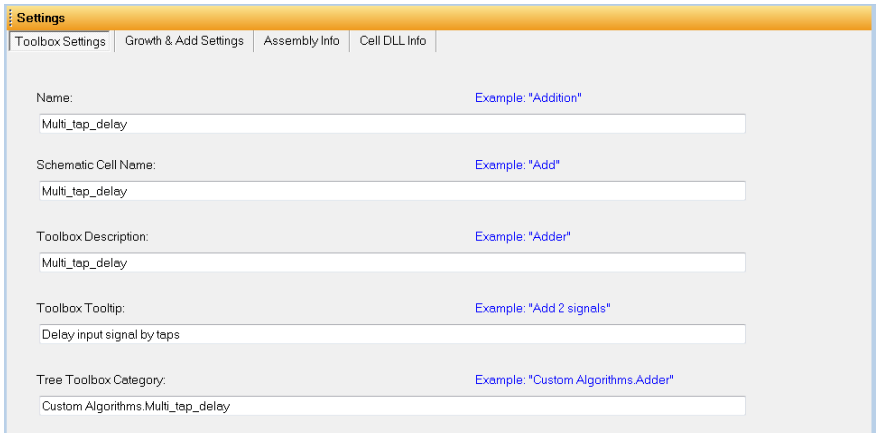



There are a few general rules for memory allocation:

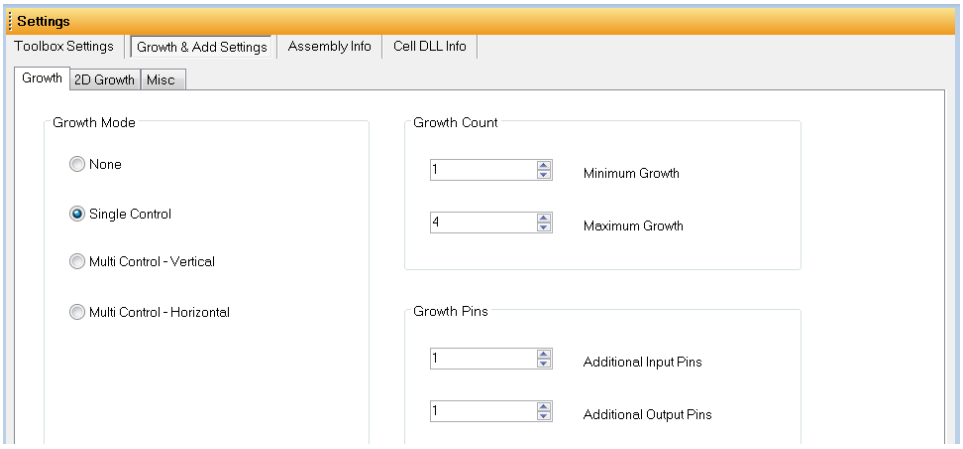
- All global variables/buffers/tables accessed from an Algorithm are shared across instances of all the Algorithms inserted in the schematic.
- Code and data can be placed in any section.

- Only extended precision state memory is supported. Tables or parameters cannot be in extended precision.
- Extended precision access must be cleared at the beginning and restored before exiting if the Plug-In uses either State B or State C memory buffers. This is because Applications may place the 32-bit state memory buffers in the same block as the Extended Precision state memory buffer.
- Since the input buffers of a Plug-In can be used, with the help of a T-connector, by many subsequent Modules in a Schematic, care should be taken to preserve the input buffers and to avoid overwriting of these buffers.
- If there are 2 functions with the same name defined in a single file, the linker gives a "Multiply defined symbol" error when the schematic which uses the function is link-compile-downloaded. If the functions with same name are defined in different files, the linker doesn't throw any error during the schematic compilation and linking. It uses whichever function definition it finds first during linking. Hence, it is advisable to use unique function names if it is not intended for the functions to be shared across modules or with the target application.

In addition, users should provide some basic information about the custom algorithms, in the lower-right window, *Settings*, as shown below. Please note Tree Toolbox Category decides where to find the custom algorithm module in Sigma Studio Toolbox.

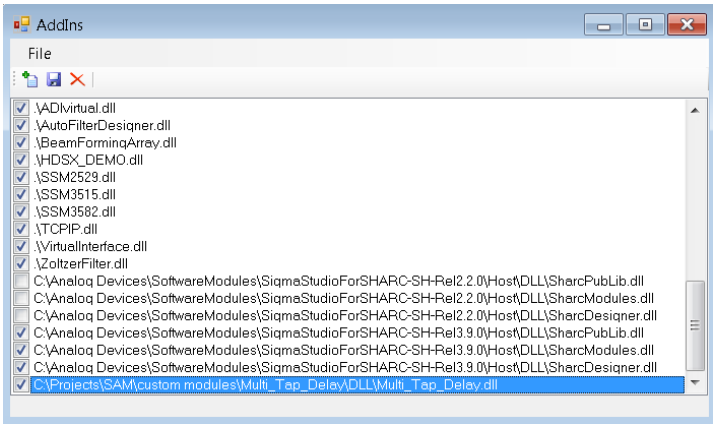



This module is created with one input channel, one delay control tab and one output channel. If we allow the module to grow more channels, we should configure it in "*Growth & Add Settings*", as shown below. The number of delay control taps and delay outputs can go up as many as four pairs or down as few as one pair.




**Step 5: Generate Dynamic Link Libraries and Load in Sigma Studio**

In the Algorithm Designer menu bar, click *Action*->*Generate Assembly*, to generate the dll file. In the Sigma Studio menu bar, click *Tools*->*Add-Ins Browser*. In the browser window, click *Add DLL* icon to add the new dll file, as shown below.




After creating a new SC-58x project in Sigma Studio, the new module could be seen in the Tree Toolbox, ADSP-SC58x, Custom Algorithms, as shown below.