CHAPTER 1

PROJECT CHANGES WHEN USING CLA FOR POWERSUITE SUPPORTED SOLUTIONS

To enable easy porting of the ISR code to CLA the following items are implemented in the software structure. Note the below applies to CLA on F28004x , F28377x and F2807x devices.

1.1 Project structure

Following image shows the typical set of files in a digital power system solution. The core algorithmic portions of the code are in the solution specific files (solution.c/h). The device specific functions are kept in the <solution>_board.c/h.

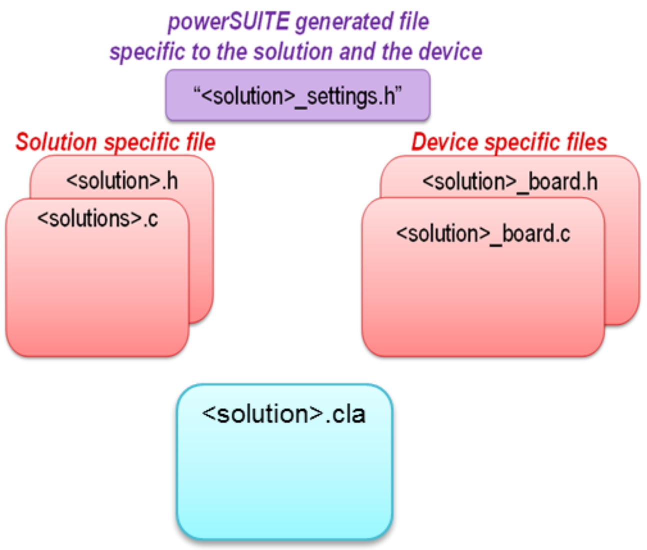
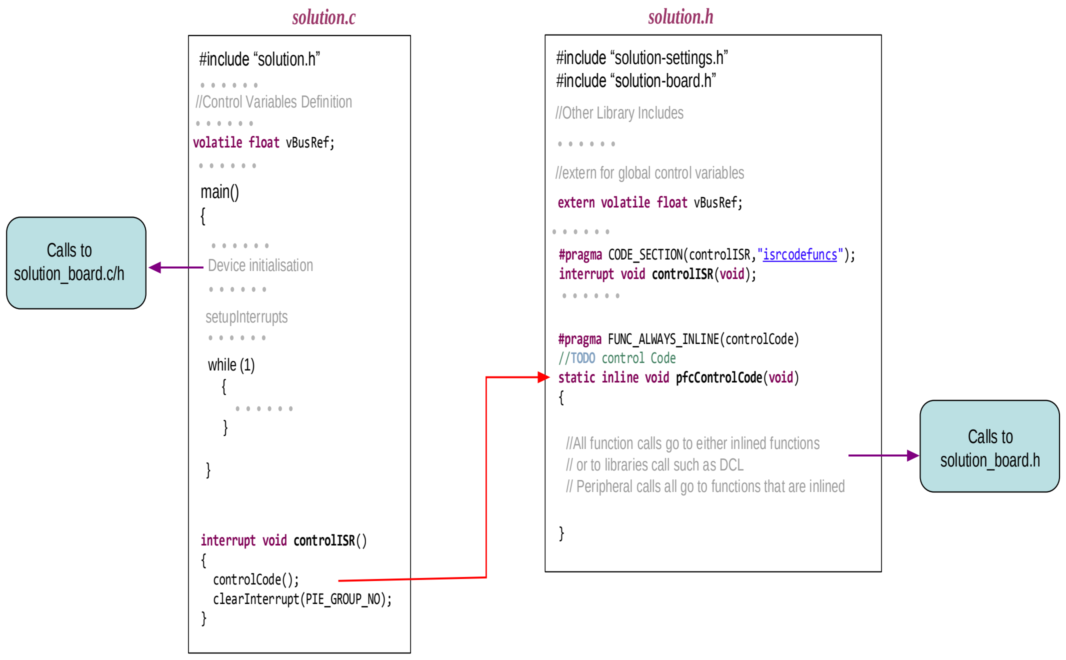


Fig. 1 Project structure of powerSUIT supported digital power solutions

Now to support CLA *.cla file is also added in the project.

1.2 Core ISR function is in a header file

To enable the ISR to run on CLA without having to re-code or copy/paste the code. The core function of the ISR is written in the header file as an inline function. This enables the function to be called from the C28x side or the CLA side, maintain the same source base. Following image shows the flow of the software when this structure is implemented on the C28x side.



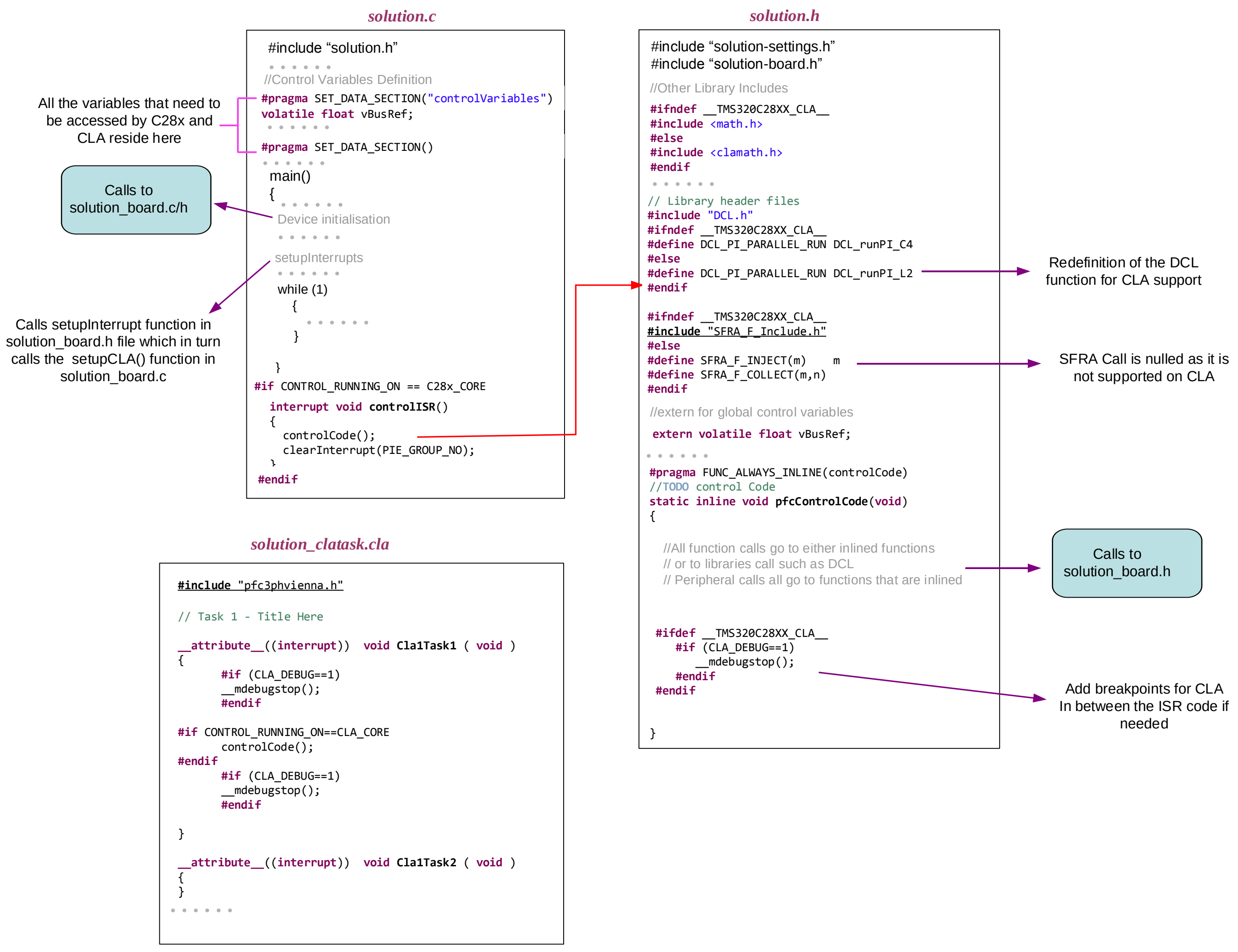
```
#include "solution.h"
 • • • • • •
//Control Variables Definition
• • • • • •
volatile float vBusRef;
 • • • • • •
 main()
 {
   • • • • • •
   Device initialisation
   • • • • • •
   setupInterrupts
   • • • • • •
   while (1)
     {
       • • • • • •
     }
 }

interrupt void controlISR()
{
  controlCode();
  clearInterrupt(PIE_GROUP_NO);
}
```

```
#include "solution-settings.h"
#include "solution-board.h"
//Other Library Includes
• • • • • •
//extern for global control variables
extern volatile float vBusRef;
• • • • • •
#pragma CODE_SECTION(controlISR,"isrcodefuncs");
interrupt void controlISR(void);
 • • • • • •

#pragma FUNC_ALWAYS_INLINE(controlCode)
//TODO control Code
static inline void pfcControlCode(void)
{

  //All function calls go to either inlined functions
  // or to libraries call such as DCL
  // Peripheral calls all go to functions that are inlined

}
```

Fig. 2 Typical code flow of a digital power solution when running on C28x CPU

1.3 Change for CLA

Fig. 3 shows the code flow when the solution is run on the CLA. The changes required when moving to the CLA are elaborated upon in the following sections.



All the variables that need to be accessed by C28x and CLA reside here
Calls to solution_board.c/h
Calls setupInterrupt function in solution_board.h file which in turn calls the setupCLA() function in solution_board.c
Redefinition of the DCL function for CLA support
SFRA Call is nulled as it is not supported on CLA
Calls to solution_board.h
Add breakpoints for CLA In between the ISR code if needed

solution.c

```
#include “solution.h”
 • • • • • •
//Control Variables Definition
#pragma SET_DATA_SECTION("controlVariables")
volatile float vBusRef;
 • • • • • •
#pragma SET_DATA_SECTION()
• • • • • •
 main()
 {  • • • • • •
   Device initialisation
   • • • • • •
  setupInterrupts
   • • • • • •
   while (1)
     {
        • • • • • •
     }

 }
#if CONTROL_RUNNING_ON == C28x_CORE
  interrupt void controlISR()
  {
    controlCode();
    clearInterrupt(PIE_GROUP_NO);
  }
 #endif
```

solution.h

```
#include “solution-settings.h”
#include “solution-board.h”
//Other Library Includes
#ifndef __TMS320C28XX_CLA__
#include <math.h>
#else
#include <clamath.h>
#endif
 • • • • • •
// Library header files
#include "DCL.h"
#ifndef __TMS320C28XX_CLA__
#define DCL_PI_PARALLEL_RUN DCL_runPI_C4
#else
#define DCL_PI_PARALLEL_RUN DCL_runPI_L2
#endif

#ifndef __TMS320C28XX_CLA__
#include "SFRA_F_Include.h"
#else
#define SFRA_F_INJECT(m)    m
#define SFRA_F_COLLECT(m,n)
#endif
//extern for global control variables
 extern volatile float vBusRef;
• • • • • •
#pragma FUNC_ALWAYS_INLINE(controlCode)
//TODO control Code
static inline void pfcControlCode(void)
{

   //All function calls go to either inlined functions
   // or to libraries call such as DCL
   // Peripheral calls all go to functions that are inlined


 #ifdef __TMS320C28XX_CLA__
    #if (CLA_DEBUG==1)
       __mdebugstop();
    #endif
 #endif

}
```

solution_clatask.cla

```
#include "pfc3phvienna.h"

// Task 1 - Title Here

__attribute__((interrupt))  void Cla1Task1 ( void )
{
       #if (CLA_DEBUG==1)
       __mdebugstop();
       #endif

#if CONTROL_RUNNING_ON==CLA_CORE
       controlCode();
#endif
       #if (CLA_DEBUG==1)
       __mdebugstop();
       #endif

}

__attribute__((interrupt))  void Cla1Task2 ( void )
{
}
• • • • • •
```

Fig. 3 Code flow when solution ported to CLA, changes spiked out

1.3.1 Changes to <solution-settings.h>

A new #define is created to tell whether the intention is to run the code on the C28x or the CLA, for example #define CONTROL_RUNNING_ON . If the option to select the core to run the control loop is available on the powerSUITE page this variable will automatically be set appropriately. It is used to implement some specific tasks that differentiate the ISR code running on C28x or CLA.

```
#define C28x_CORE 1
#define CLA_CORE 2
#define CONTROL_RUNNING_ON 1
```

1.3.2 Changes to <solution.c>

a. First of all, the control variables must be placed in a CLA accessible memory, this is achieved using **#pragma SET_DATA_SECTION** as shown in Project structure diagram for CLA below. On the F28377D and F28004x when data memory is assigned to the CLA both C28x and CLA can read and write to the memory. This enabled very flexible way of offloading tasks to CLA without worrying about memory access protection.

```
#pragma SET_DATA_SECTION(“controlVariables”)
float var1;
int32_t flag1;
….
#pragma SET_DATA_SECTION()
```

b. The data types are different on the CLA, and hence for the integer flags and variables, it’s best to use 32 bit type. Make sure the variables used by CLA are 32 bit int, or float (32 bit). This will avoid any un-necessary issues when compiling code on the CLA.

c. Setup the CLA, for this as part of the setup interrupt function a setup CLA function is called and memory assignment and copy of code to the CLA is performed in this section. By default when the code is compiled on the CLA the code in compiled into the Cla1Prog function. Refer to the CMD file and CLA compiler information. The assignment for task trigger is also performed in this function.

d. The controlISR interrupt function is guarded by a new #define by which we can tell whether the code is running on the C28x or the CLA. This prevents unnecessary program memory usage when the code is run on the CLA, otherwise the same code is compiled for C28x and CLA when running the solution on CLA.

```
#if CONTROL_RUNNING_ON == C28x_CORE

    interrupt void controlISR(void)
    {
        pfcControlCode();
        clearInterrupt(C28x_CONTROLISR_INTERRUPT_PIE_GROUP_NO);
    }

#endif
```

1.3.3 Changes to <solution.h>

a. Certain libraries may have differences on CLA, for example the <math.h> is unavailable on the CLA. For this guard defines need to be added and an appropriate include added. To support compilation of the same code on CLA and C28x, re-defines for the routines in the math.h file for CLA are added to this file as shown below.

```
#ifndef __TMS320C28XX_CLA__
#include <math.h>
#else
#define sinf  CLAsin
#define cosf  CLAcos
#define sqrtf CLAsqrt
#define __einvf32 __meinvf32
#define __eisqrtf32 __meisqrtf32
#include <clamath.h>
#endif
```

b. Similarly, certain libraries have an equivalent function for the CLA and hence they must be redefined for running on the CLA. Or if they do not have any equivalents the calls nulled. Below is an example of two such common functions, SFRA and DCL lib.

```
…
#ifndef __TMS320C28XX_CLA__
#include "SFRA_F_Include.h"
#else
#define SFRA_F_INJECT(m)    m
#define SFRA_F_COLLECT(m,n)
#endif
….
#include "DCL.h"
#ifndef __TMS320C28XX_CLA__
```

```
#define DCL_PI_PARALLEL_RUN DCL_runPI_C4
#else
#define DCL_PI_PARALLEL_RUN DCL_runPI_L2
#endif

….
```

c. Optionally one can also add CLA breakpoints in the middle of the ISR code as needed.

1.3.4 Adding <solution.cla> file

The solution.cla file is fairly straight forward, just including the solution.h header file and calling the controlCode function in the solution.h file. The call to that function is guarded by the #define CONTROL_RUNNING_ON to prevent un-necessary memory usage when the code is not run on the CLA.

1.3.5 Changes to *.CMD file, and device.c/h

As shown in the previous sections, isrcode is mapped into a section called “isrcodefuncs”. This section is placed in the same memory as the CLA program space. “dclfuncs” are also assigned to this block as well using a “GROUP” as shown below in the CMD file.

```
GROUP
    {
        isrcodefuncs
        dclfuncs
    }    LOAD = FLASHH,
         RUN =  RAMLS2LS3LS4LS5,
         LOAD_START(_isrcodefuncsLoadStart),
         LOAD_SIZE(_isrcodefuncsLoadSize),
         LOAD_END(_isrcodefuncsLoadEnd),
         RUN_START(_isrcodefuncsRunStart),
         RUN_SIZE(_isrcodefuncsRunSize),
         RUN_END(_isrcodefuncsRunEnd),
         PAGE = 0, ALIGN(4)

/* CLA specific sections */
Cla1Prog : LOAD = FLASHH,
           RUN = RAMLS2LS3LS4LS5,
           LOAD_START(_Cla1ProgLoadStart),
           RUN_START(_Cla1ProgRunStart),
           LOAD_SIZE(_Cla1ProgLoadSize),
           PAGE = 0, ALIGN(4)
```

To accommodate for this change in the CMD file. Device_init() function in “device.c” file needs to incorporate an additional memory copy which is for the section “isrcodefuncs”.

“divice.h”

```
//Add externs for the memory section , these are defined in the CMD file
extern uint16_t isrcodefuncsLoadStart;
extern uint16_t isrcodefuncsLoadEnd;
extern uint16_t isrcodefuncsLoadSize;
extern uint16_t isrcodefuncsRunStart;
extern uint16_t isrcodefuncsRunEnd;
extern uint16_t isrcodefuncsRunSize;
```

“divice.c”

```
 void Device_init(void)
{

….
    #ifdef _FLASH
    // now copy any additional code that needs to be copied from flash to ram
    memcpy(&isrcodefuncsRunStart, &isrcodefuncsLoadStart,
           (size_t)&isrcodefuncsLoadSize);
    #endif

…..

}
```

1.3.6 Changes to <solution-board.c/h>

a. Add guards around the declaration of the interrupts and add CLA task defines:

```
//CLA C Tasks defined in Cla1Tasks_C.cla
__attribute__((interrupt))  void Cla1Task1();
__attribute__((interrupt))  void Cla1Task2();
__attribute__((interrupt))  void Cla1Task3();
__attribute__((interrupt))  void Cla1Task4();
__attribute__((interrupt))  void Cla1Task5();
__attribute__((interrupt))  void Cla1Task6();
__attribute__((interrupt))  void Cla1Task7();

extern uint16_t Cla1ProgLoadStart;
extern uint16_t Cla1ProgLoadEnd;
extern uint16_t Cla1ProgLoadSize;
extern uint16_t Cla1ProgRunStart;
extern uint16_t Cla1ProgRunEnd;
extern uint16_t Cla1ProgRunSize;
```

```
// ISR related
#if CONTROL_RUNNING_ON == C28x_CORE

#ifndef __TMS320C28XX_CLA__
    #pragma INTERRUPT (controlISR, HPI)
    #pragma CODE_SECTION(controlISR,"isrcodefuncs");
    interrupt void controlISR(void);
#endif

#else
#endif
```

b. Call the setupCLA function as part of the setupInterrupt() function. One can also put the setup of the C28x interrupt guarded by #define CONTROL_RUNNING_ON, this prevents a redundant interrupt being registered on the C28x.

```
static inline void setupInterrupt(void)
{
    …..
#if CONTROL_RUNNING_ON == C28x_CORE
    // PWM was already enabled to trigger ISR
    Interrupt_register(C28x_CONTROLISR_INTERRUPT, &controlISR);
    clearInterrupt(C28x_CONTROLISR_INTERRUPT_PIE_GROUP_NO);
    Interrupt_enable(C28x_CONTROLISR_INTERRUPT);
#endif

#if (CONTROL_RUNNING_ON==CLA_CORE)
    setupCLA();
#endif
        ….
}
```

c. Optionally, based on the warnings you may want to suppress some warnings that are a result on type differences between the C28x and the CLA. Below is an example

```
static inline void resetProfilingGPIO2(void)
{
    #pragma diag_suppress=770
    #pragma diag_suppress=173
    HWREG(GPIODATA_BASE  + GPIO_O_GPACLEAR ) = GPIO_PROFILING2_CLEAR;
    #pragma diag_warning=770
    #pragma diag_warning=173
}
```

d. For GPIO the master needs to be defined, hence if CLA needs to control the GPIO the following piece of code must be added.

```
void setupProfilingGPIO(void)
{
    ….

#if CONTROL_RUNNING_ON == CLA_CORE
    GPIO_setMasterCore(GPIO_PROFILING1, GPIO_CORE_CPU1_CLA1);
#endif
}
```

e. Finally , create a new setupCLA function

```
void setupCLA(void)
{

#if CONTROL_RUNNING_ON==CLA_CORE
        // copy code from FLASH to CLA accessible memories CLA
    memcpy((uint32_t *)&Cla1ProgRunStart, (uint32_t *)&Cla1ProgLoadStart,
            (uint32_t)&Cla1ProgLoadSize );

    // Assign memory to CLA and set memory type to Program or Data
    // Assign LS0 -LS5 to CLA
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS0,MEMCFG_LSRAMMASTER_CPU_CLA1);
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS1,MEMCFG_LSRAMMASTER_CPU_CLA1);
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS2,MEMCFG_LSRAMMASTER_CPU_CLA1);
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS3,MEMCFG_LSRAMMASTER_CPU_CLA1);
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS4,MEMCFG_LSRAMMASTER_CPU_CLA1);
    MemCfg_setLSRAMMasterSel(MEMCFG_SECT_LS5,MEMCFG_LSRAMMASTER_CPU_CLA1);
    // Assign LS0 -LS2 to CLA Data, LS2-LS5 as CLA program
    MemCfg_setCLAMemType(MEMCFG_SECT_LS0,MEMCFG_CLA_MEM_DATA);
    MemCfg_setCLAMemType(MEMCFG_SECT_LS1,MEMCFG_CLA_MEM_DATA);
    MemCfg_setCLAMemType(MEMCFG_SECT_LS2,MEMCFG_CLA_MEM_PROGRAM);
    MemCfg_setCLAMemType(MEMCFG_SECT_LS3,MEMCFG_CLA_MEM_PROGRAM);
    MemCfg_setCLAMemType(MEMCFG_SECT_LS4,MEMCFG_CLA_MEM_PROGRAM);
    MemCfg_setCLAMemType(MEMCFG_SECT_LS5,MEMCFG_CLA_MEM_PROGRAM);

    // Map CLA task vectors
    #pragma diag_suppress=770

    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_1, (uint16_t)&Cla1Task1);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_2, (uint16_t)&Cla1Task2);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_3, (uint16_t)&Cla1Task3);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_4, (uint16_t)&Cla1Task4);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_5, (uint16_t)&Cla1Task5);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_6, (uint16_t)&Cla1Task6);
    CLA_mapTaskVector(CLA1_BASE , CLA_MVECT_7, (uint16_t)&Cla1Task7);

    #pragma diag_warning=770

    CLA_enableIACK(CLA1_BASE);
    CLA_enableTasks(CLA1_BASE,CLA_TASKFLAG_ALL);
    clearPWMInterruptFlag(C28x_CONTROLISR_INTERRUPT_TRIG_PWM_BASE);
    CLA_setTriggerSource(CLA_TASK_1, CLA_CONTROLISR_TRIG);

#endif

}
```

1.3.7 Using background task

The F28004x CLA supports a background task from which a CLA task can nest from, this enables offloading two ISRs to the CLA. For example a 50kHz fast current loop ISR and a 10kHz voltage loop ISR.

To enable background task for the slower 10kHz ISR, follow the steps outlined in the previous section to:

1. Define a new #define that will describe whether this ISR will run on the CLA or the C28x. If this is powerSUITE project this might already be set for you, if the CLA selection option is enabled from the CFG GUI.

   #define INSTRUMENTATION_ISR_RUNNING_ON CLA_CORE

2. Make sure the variables accessed by this second ISR are also in a CLA accessible memory. Make the necessary changes for this ISR as outlined in previous section for <solution.c> file

3. The CLA background task does not support any type of function calls to be coded in the background task. It can only nest to the CLA Task. Hence the code in the CLA background task must be completely flat. Hence the modules used by the algorithm in the CLA background task must be in-lined.

a. CLAmath: this affects the use of CLAmath lib as any call to the CLAmath will result in a function call, which will not be supported in the background task. To mitigate this some inline CLAmath functions are planned to be released in the CLAmath lib. In the meantime one can use the CLAmath header file which has the inline function and is part of the solution package itself. Also the redefinitions of the "math.h" routines on CLA must be done to the inline routines. (This might increase the CLA cycles for the faster ISR, as now the functions are inline C and not optimized assembly, however the difference is not going to be significant in most cases to affect the system performance)

```
#ifndef __TMS320C28XX_CLA__
#include <math.h>
#else
#define sinf  CLAsin_inline
#define cosf  CLAcos_inline
#define sqrtf CLAsqrt_inline
#define __einvf32 __meinvf32
#define __eisqrtf32 __meisqrtf32
#include <clamath.h>
#endif
```

b. DCL Library, DCL library includes assembly routines that are optimized such as DCL_PI_C4 for the C28x and DCL_PI_L2 for the CLA. Redefinitions for the same were added to the "solution.h" file as outlined in the previous sections. If the calls to this routine needs to happen from the background task one must use the inline version of the function i.e. DCL_PI_C5. As these are inline functions, no re-definitions are needed for these modules.

4. solution.cla file must be changed to accommodate the background task

```
__attribute__((interrupt("background")))  void Cla1BackgroundTask ( void )
{
```

```
…
#if INSTRUMENTATION_ISR_RUNNING_ON ==CLA_CORE
    instrumentationCode();
#endif
 ….
}
```

5. Changes must be made to the solution-board.c file to accommodate settings for the background task in the setupCLA function:

```
    ….
      CLA_enableHardwareTrigger(CLA1_BASE);
      CLA_setTriggerSource(CLA_TASK_8, CLA_INSTRUMENTATIONISR_TRIG);
      CLA_enableBackgroundTask(CLA1_BASE);
```

6. Make sure the cla background task support is enabled, refer to the C28x Compiler guide for instructions