Application Note

PRU - Getting started Guide (RTOS) - AM243x LP EVM / AM64x GP EVM

Rev. <ref>
<date>

Revision History

Version	Date	Author	Description
0.1	Sept 28 - 2022	Thomas Leyrer	Initial version using Code Composer Studio (CCS) for build and debug, AM243x Launchpad
0.2	Oct 18 – 2022		Added OSPI boot image

Table of Contents

1 Introduction

Programable Real-time Unit (PRU) is a 32 bit non-pipelined RISC CPU which solves interface and processing functions with minimum latency and minimum jitter. In addition, it is used to implement custom protocols over standard interfaces or add missing standard interfaces on Sitara MCU+ and MPU products. Besides non-pipelined CPU at a speed of up to 333 MHz there are two key differentiating features.

- GPIO signals are directly mapped to internal register file
- Broadside extension which supports 1024 bit data bus for data transfer and data processing accelerators

This document limits the scope to direct GPIO and simple broadside accelerator functions. In total AM243x and AM64x devices contain 12 PRU cores which can work independent from each other or fully synchronized as multi-core real-time domain. The cores are split out into two Industrial Communication Subsystem (ICSS_G0 and ICSS_G1). Table-1 shows a summary of all cores with memory configuration and global address space of instruction memory. All cores have together 96kB of zero wait-state instruction memory. When PRU is halted, new code can be downloaded which is useful in case program memory is limited. For example, configuration and initialization code is executed first before operational code is downloaded and executed.

ICSS_G instance	PRU instance	IMEM [kBytes]	global address IMEM
PRG0	PRU0	12	0x03003 4000
	RTU_PRU0	8	0x03000 4000
	TX_PRU0	6	0x03000 A000
	PRU1	12	0x30003 8000
	RTU_PRU1	8	0x03000 6000
	TX_PRU1	6	0x03000 C000
PRG1	PRU0	12	0x0300B 4000
	RTU_PRU0	8	0x03008 4000
	TX_PRU0	6	0x03008 A000
	PRU1	12	0x3000B 8000
	RTU_PRU1	8	0x03008 6000
	TX_PRU1	6	0x03008 C000

Table 1 – PRU cores and instruction memory

Control of PRU operation is provided through ICSSG_PRU_CONTROL register. It provides ARM core the possibility to start, stop and reset the core. When PRU core is stopped then ARM can download new code in instruction memory. There is a control bit which enables PRU cycle counter and stall counter. These counters can be read by PRU during runtime and while working with the debugger. Description of the registers can be found in TRM chapter 6.4.14.1. The debugger supports control register view.

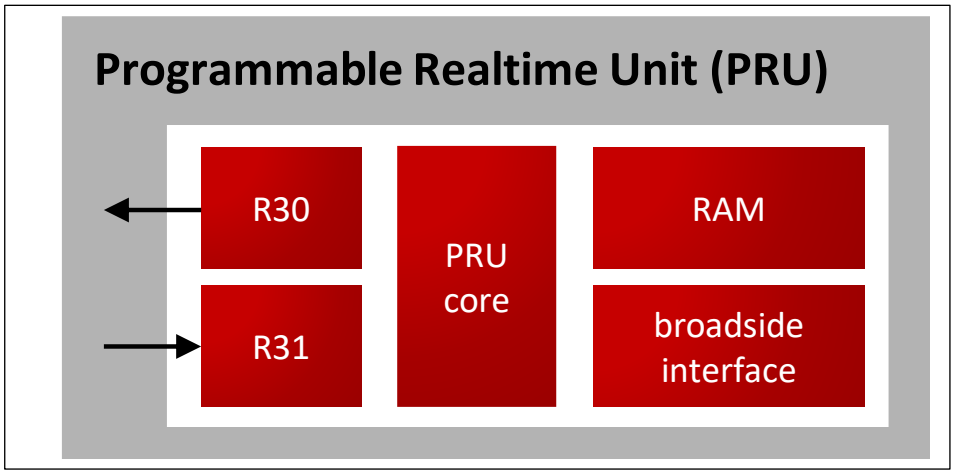


Figure 1- Programmable Real-time Unit

PRU core has 32 register (R0-R31) each with 32 bit width. There are 32 bit constant registers (C0-C31) which have predefined address of commonly used IOs and memory. A few of the constant registers can be programmed with certain offset. These registers are used in combination with load and store instructions to save on normal registers and instruction memory.

Figure 1 shows simplified block diagram of PRU. R30 has 20 GPOs directly mapped to bits 0 - bit 19. R31 has 20 GPIs mapped the lower 20 bits of the register. These registers can be programmed at a bit level, with 8/16/32 bit instructions. For example “set r30, r30, 3” instruction sets external pin PRU0_GPO3 to logic high – typically 3.3V level. Clear instruction sets the GPO to GND. R30 register can also be part of any instruction which operates on register file. For example instruction “xor r31.b1, r30.b0, r30.b0” inverts signals on GPI0..7 to GPO8..15 in 3 ns. Note that GPIs and GPOs are mapped to the same pins and you need to select which direction to use. For fast direction switch such as a data bus of parallel interface, there is common output enable register inside ICSS – see TRM “6.4.2.2 PRU_ICSSG Fast GPIO pins” for details. R30 and R31 have additional function for event generation and polling which is not described in this document. PRU polling for internal and external events can be replaced now (ICSS_G devices) with real-time task manager. The task manger support two cycle interrupt latency with no jitter and belongs to the broadside extension of PRU. There are broadside functions for data processing and data transfers. A summary chart of broadside functions is shown in the appendix. It lists the registers used by the XIN, XOUT instructions (broadside instructions) which is important to know as they may overlap with register usage with own program or C compiler.

2 Software Installation

Installation of various tools from ti.com website is required to get started with PRU firmware projects. This chapter describes all steps needed to develop, run and debug PRU code.

2.1 MCU+ SDK for AM243x/AM64x

MCU plus software development kit (SDK) contains sample projects for PRU functions and driver/APIs for managing PRU from ARM side. Install the latest from following link:
AM64x: https://www.ti.com/tool/PROCESSOR-SDK-AM64X
AM243x: https://www.ti.com/tool/MCU-PLUS-SDK-AM243X
Default installation path c:/ti and you will find MCU+ SDK under this directory. There is a README_FIRST_AM243X.html file which can be opened for off-line documentation.
The download instruction [link] also ask for installing SYSCONFIG
https://www.ti.com/tool/SYSCONFIG

Python 3
https://www.python.org/downloads/windows/

PRU compiler
https://www.ti.com/tool/PRU-CGT

TI CLANG Compiler Toolchain
https://software-dl.ti.com/codegen/esd/cgt_public_sw/ARM_LLVM/1.3.1.LTS/ti_cgt_armllvm_1.3.1.LTS_windows-x64_installer.exe

Other components like OpenSSL and Mono Runtime are not required for Windows based system

2.2 Code Composer Studio (CCS)

There can be various options with the PC or notebook which are important to understand. In case CCS is not installed before on the PC then download latest version of CCS from ti.com with following link:
New Install: https://www.ti.com/tool/CCSTUDIO
During installation process the tool asks for custom installation which is recommended. Make sure to pick one of the Sitara MCU or MPU components as shown below.

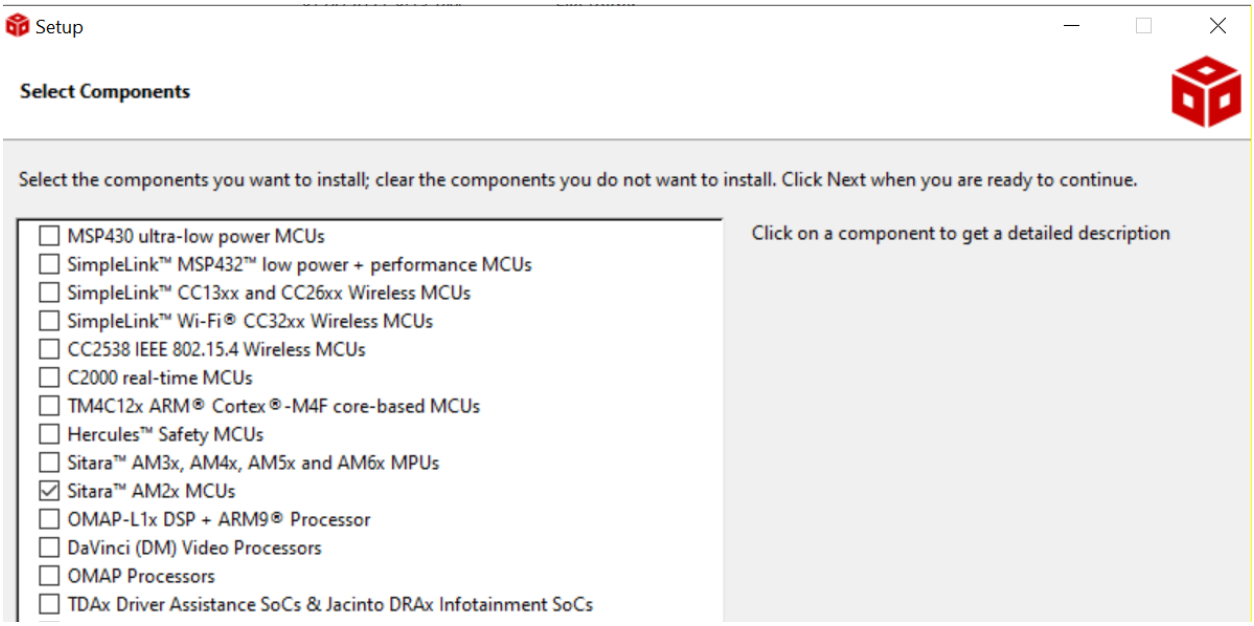


Figure 2 – Sitara Component Selection

In case there is an older version of CCS installed it does not support latest ARM CLANG compiler and PRU compiler. It is recommended to upgrade CCS versions older than version 10 to the latest. Third option is that there is CCS for MCUs such as MSP430 and C2000 CPUs installed. In this case you will lack Sitara device components, the compiler for ARM and PRU. New compilers are installed from CCS HELP Menu -> Install Code Generation Compiler Tools. After CCS is installed, check that all required components are installed.
CCS “Window Menu -> Preferences” gives an overview of installed compiler and products.
Check that compilers for ARM TI Clang and PRU are shown in Discovered tools window.

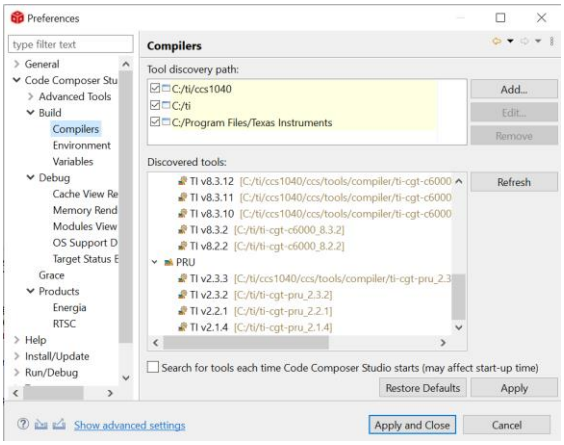


Figure 3- CCS Compiler selection

Under CCS “Window -> Preferences” Code Composer Studio -> Products you should find MCU+ SDK and SYCONFIG.

2.3 EVM software setup

Next step in bringing up the EVM for operation is bootloader configuration. Follow the steps in the [link] which configures UART for terminal operation, download boot loader into flash and allows for no boot mode. The examples in this document run with all boot modes including NO BOOT MODE.

2.4 Target Configuration File

In the description of creating target configuration file for the EVM the PRU cores are disabled. Do not follow this step and make sure PRU cores of ICSS_G0 and ICSS_G1 are enabled. To get to the settings of connections you need to click on the advanced tab at the bottom of the screen. Figure 4 shows the PRU cores enabled.

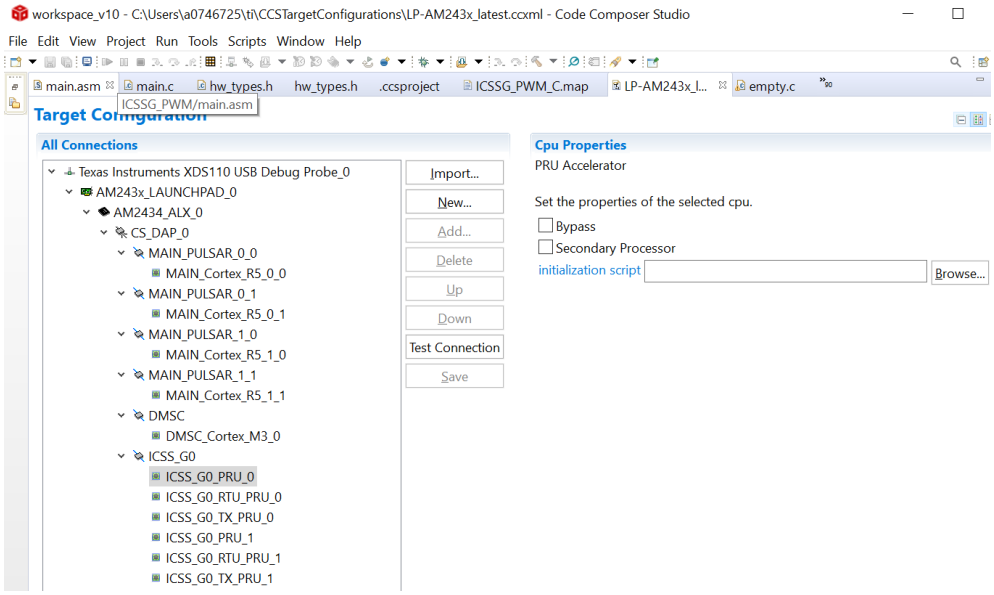


Figure 4 – Target Configuration file with PRU support

After CCS load and power-up of the EVM the first step is to connect to target configuration file.

2.5 Start-up script for no boot mode

In no boot mode requires to run start-up script on scripting console. Bring up scripting console using Menu VIEW -> Scripting Console. At the command prompt enter the start-up script

```
js:> loadJSFile "C:/ti/mcu_plus_sdk_am243x_08_03_00_18/tools/ccs_load/am243x/load_dmsc.js"
```

For different EVM and sdk replace the SDK device name and revision:

```
js:> loadJSFile "C:/ti/mcu_plus_sdk_am64x_08_01_00_36/tools/ccs_load/am64x_am243x/load_dmsc.js"
```

Final step before working with PRU cores is to connect with core. A right click on the PRU core brings up a pop menu. Select connect to get PRU in suspended mode. In this mode the firmware can be loaded and started.

3 Hardware Setup

The AM243x Launchpad hardware requires power over USB Type-C connector and JTAG/UART connection over USB cable. You should first plug in power before connecting to JTAG/UART. It is important to configure the boot mode settings on die DIP switch.

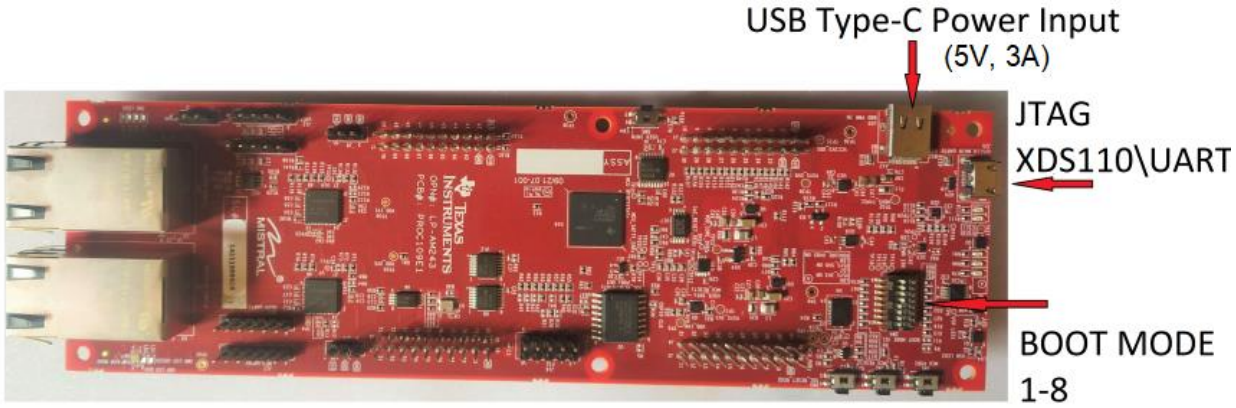
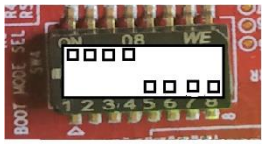
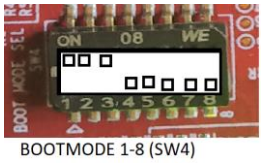


Figure 5 – AM243x Launchpad with Power and Debug connector

In case of boot loader is not stored into external flash memory the DIP-switch is set to NO BOOT MODE as shown in the image on the left. In this mode the user needs to run a start-up script on scripting console.





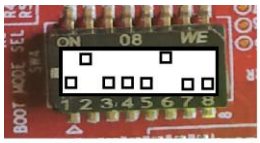


Figure 6 - boot modes

Figure 6 shows the different boot modes. With UART boot mode and python script a boot loader and application image can be flashed in to external memory. QSPI boot mode is selected to boot from external flash. The procedure is described in Getting started of MCU+ SDK.

4 Additional Help from e2e forum

E2E forum entries related to PRU:

4.1 How to use CCS to connect to PRU (ICSS_G)?

https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1045297/faq-am64x-am24x-how-to-use-code-composer-studio-ccs-to-connect-to-pru_icssg

4.2 How to check and set PRU Core Frequency in CCS?

https://e2e.ti.com/support/processors-group/processors/f/processors-forum/1041347/faq-pru_icssg-how-to-check-and-set-pru-core-frequency-in-ccs

5 Examples

Linker command files for different devices and PRU cores can be found at:
https://git.ti.com/cgit/pru-software-support-package/pru-software-support-package/tree/labs/Getting_Started_Labs/linker_cmd

For AM243x devices the AM64x cmd files can be used. The linker command file defines instruction memory, data memory and peripheral address. Section .text is entry point for both assembler and C language and needs to start at instruction memory address 0 of PRU which is the default reset program counter of PRU. There is an option to program reset vector to different address in PRU configuration register. For C language _c_int00* is also mapped to address 0. Additional data memory like stack pointer and .bss section resides in 8kB data memory.

5.1 My first PRU program – C language

Learning goal:

- Shows basic examples how to generate PRU C Program
- Example introduces linker command file and map file
- Example shows CCS debug window including disassembler window
- Example explains PRU register usage on function calls

There are two options to generate CCS projects. With MENU File -> Import an existing CCS project can be loaded. A new CCS Project is generated using MENU File -> New -> Project -> Code Composer Studio. It brings up the New Project wizard as shown in figure 7.

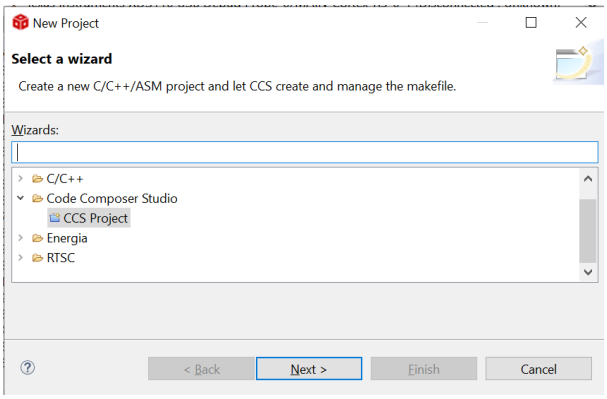


Figure 7- Generate new CCS Project

Select target hardware like AM243x Launchapd, PRU core, project name and empty project (main.c) according to figure 8.

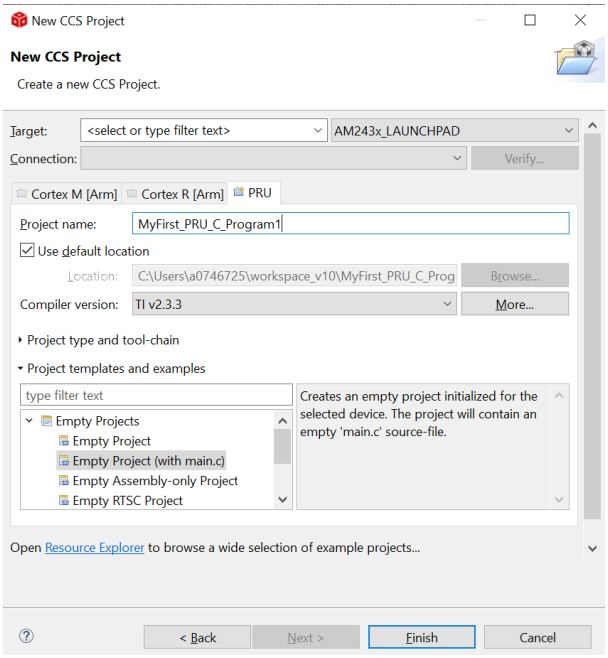


Figure 8 - Generate PRU C Project

A new project is generated which is listed in the Project Explorer in CCS. To view the properties of the project, click on the project and press [ALT+ENTER] – alternative is right-click with the mouse on the project name and select Properties from the pop-up menu. The CCS General selector shown in figure 9 gives an overview of which device and tools are selected. The linker command file should be selected to pick PRU memory map which is different for PRU, RTU_PRU or TX_PRU. In the example below there is AM64x_PRU0.cmd selected which also fits for AM243x devices as there is equal PRU_ICSS_G subsystem on both devices.

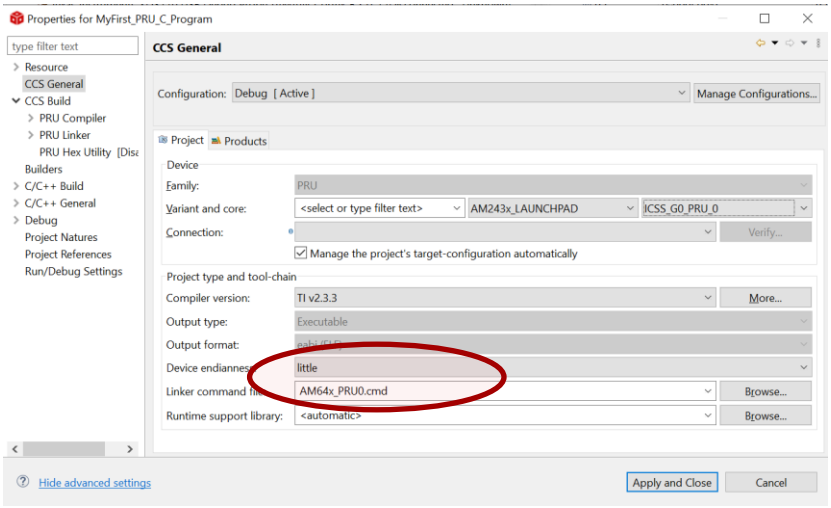


Figure 9 - Project configuration

The example source code of main.c is in the appendix. It shows basic operations of main and function call with variables on stack and variables mapped to PRU data memory. The linker command file sets instruction and data

memory. Stack size can be configured via project properties – CCS Build – PRU Linker - Basic Options. After New project is generated the empty main.c can be replaced by the example in appendix. The build process starts with either right-click on project and Build Project or using the “hammer” icon on the menu bar. After build has finished without errors the .out file is loaded through MENU -> RUN -> Load -> Load Program. Browse for the debug folder on the project to select the out file. Make sure you have selected PRU core when loading PRU out file. Next step is to run the code with either single step debug (F5-key) or free-run (F8-key). MENU – View -> disassembly brings up the window as shown in figure 10. The disassembler show C instructions and assembler instructions interleaved.

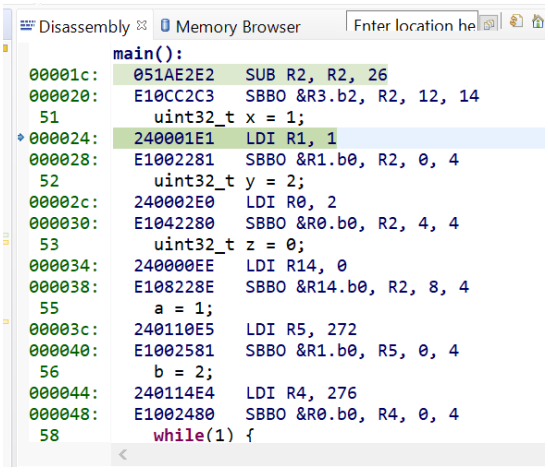


Figure 10 – Disassembly window

As described in the source code, the C function call uses register R14 and R15 for function arguments and return value. Figure 11 shows the multiple windows of the debugger. In debug tab all cores are shown and the selected core is in suspended mode – see target configuration section. On the right side there are multiple tabs to view variables, expressions, PRU registers and breakpoints. Bottom left side shows the C source code and current position of program counter is indicated by an arrow on the left. Disassembly and memory tab are shown on the bottom right side. For PRU data memory choose PRU_Device_Memory in Memory Browser tab.

The build process generates a map file which is stored in debug folder of the current project. The map file lists all the memory sections used by the program. For bigger projects it is useful to see how much instruction memory is used by the program.

Extract from map file:

```
MEMORY CONFIGURATION

         name            origin    length      used     unused   attr    fill
----------------------  --------  ---------  --------  --------  ----  --------
PAGE 0:
  PRU_IMEM              00000000   00003000  000000d4  00002f2c  RWIX

PAGE 1:
  PRU0_DMEM_0           00000000   00001000  00000100  00000f00  RWIX
```

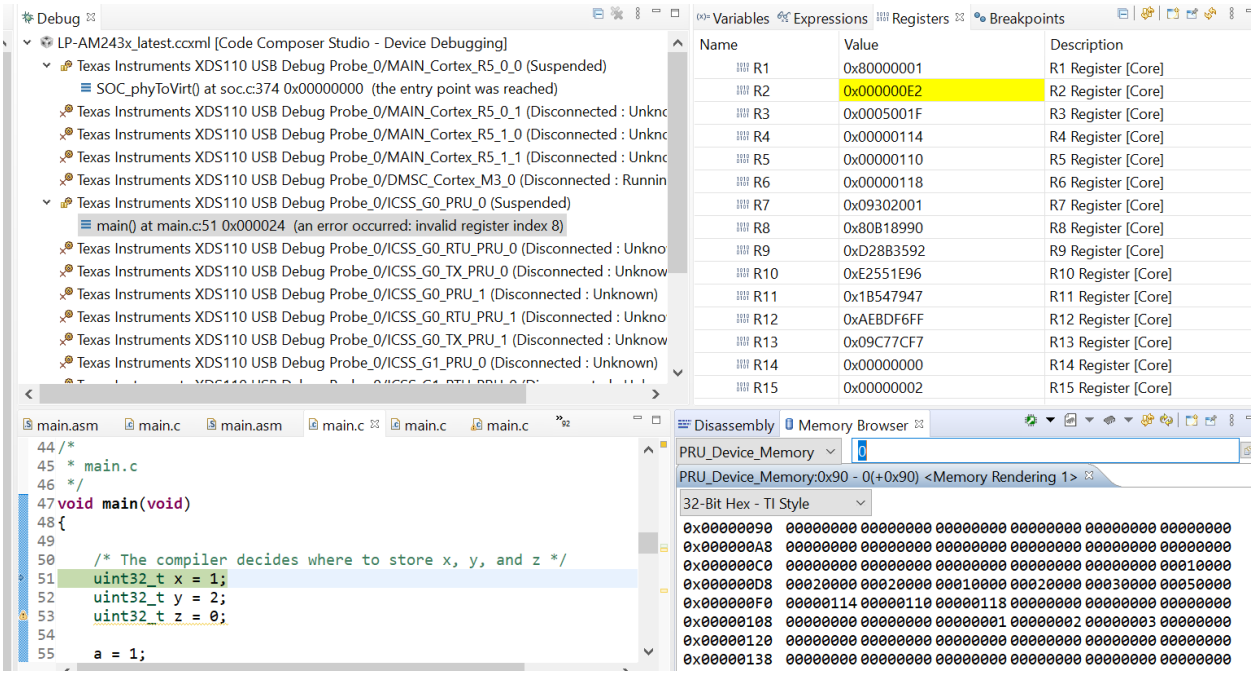


Figure 11 - CCS Windows for PRU debug

5.2 My first PRU program – Assembler

Learning goal:

- Example shows PRU assembler instructions with register view
- Example shows limits on addressing 1 bit, 8 bits, 16 bits and 32 bits
- Example shows how to use PRU cycle counter
- Example shows 'move to line' and manual change of PRU register to change program flow

The advantage of PRU assembler program is deterministic real-time performance. Every instruction on internal register file and broadside acceleration is single cycle. At PRU speed at 333 MHz one cycle is 3 ns. Especially usage of broadside accelerator overlaps with C compiler stack pointer register R2 and arguments R14-29. In this case user can program in assembler or mix assembler with C Code.
Below main.asm file shows basic usage of PRU assembler instruction. Assembler source files start with directives to build .out file with the linker and definition of main label as entry point of the program which gets mapped to section .txt.

main.asm

```
  .retain     ; Required for building .out with assembly file
  .retainrefs ; Required for building .out with assembly
  .global     main
  .sect    ".text"

main:
    ldi    r2.w1, 0xffff
    add    r2.b0, r2.b1, r2.b2
    lmbd   r2.b3, r2, 1
    lsl    r2.b0, r2.b3, 3
    clr    r2.b1, r2.b1, 5
    sbco   &r2.b1, c24, 3, 1
    lbco   &r2.b2, c24, 3, 1
    xin    160,&r2, 4 ; BSWAP widget
    qbbs   label_x, r2, 17
    ldi    r31, 0x20  ; interrupt 0 + enable (bit 5)
    nop
label_x:
    halt
```

The assembler program shows usage of assembler instructions with the scope of addressing at bit, byte, 16 bit and 32 bit level. For example, first instruction after main: loads a constant into register r2 word1 which is in the middle of 32 bit register. It makes a difference whether destination register is byte or word. The ADD command sums up to bytes which overruns the byte boundary. However, the destination address of ADD is a byte and therefore the result is truncated to a byte. Figure below shows the possible byte and word addressing modes. There are also bit instructions which are defined by the bit number. The CLR instruction in the example clears bit number 5 in register 2 byte 1. ZERO and FILL are instructions which can go over all 32 registers in a single cycle. After power-up all PRU registers are random and ZERO instruction can be used to clear the complete register set in a single cycle.

Figure 12 shows CCS windows for assembler source level debug. The register view has all PRU registers and Program Counter (PC). Figure 13 shows different addressing modes and figure 14 lists all PRU instructions.

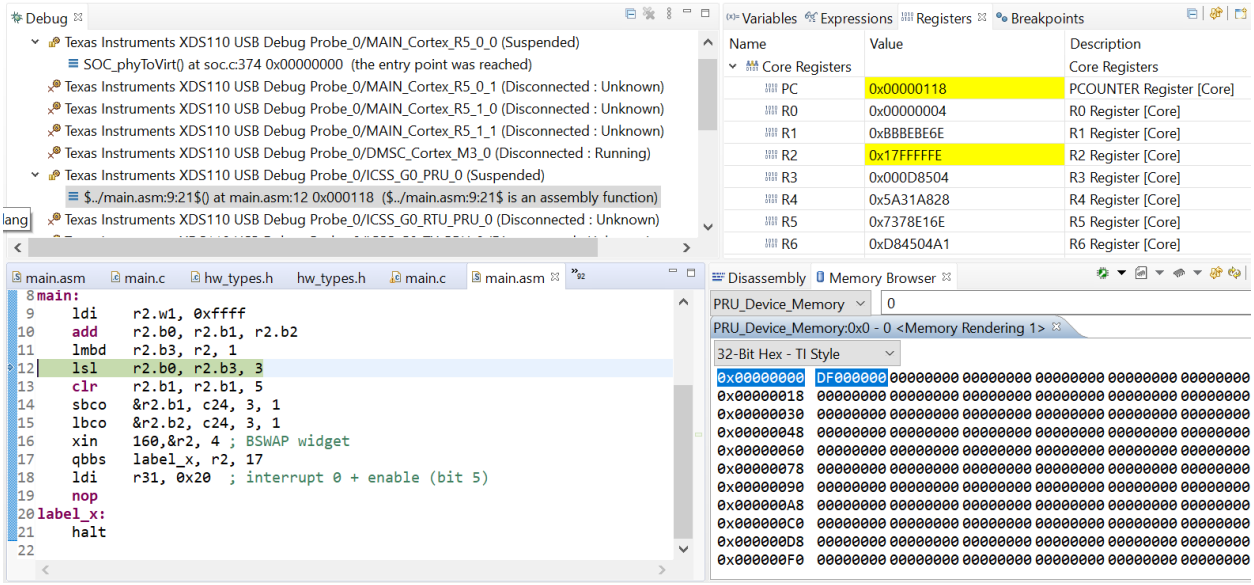


Figure 12 – CCS source level debug

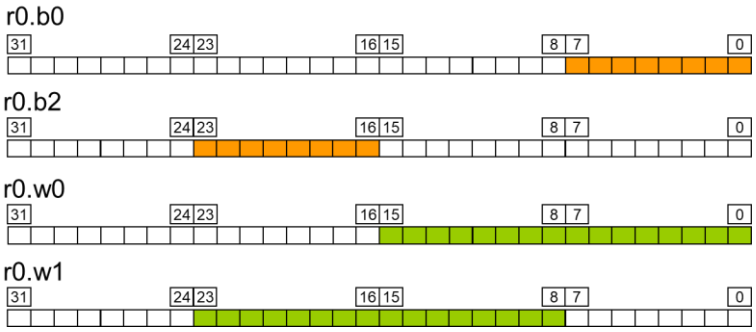


Figure 13 – Addressing for PRU Register

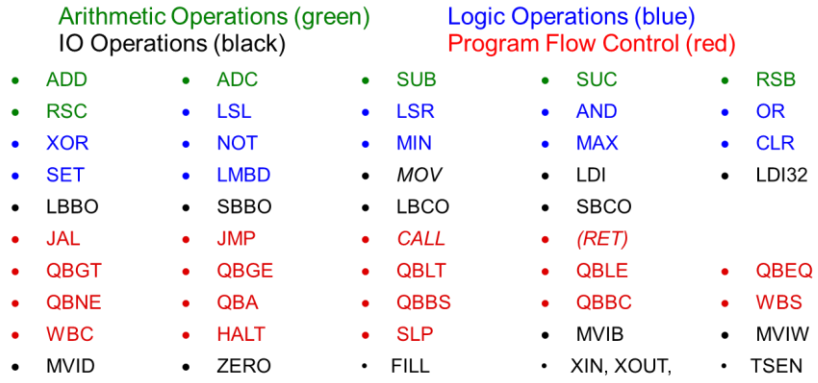


Figure 14 – PRU instruction set

In order to start new assembler project, follow the same steps as with the C project except for Empty project is now “Empty Assembly-only Project”. Figure 15 shows the selection for empty assembler project.

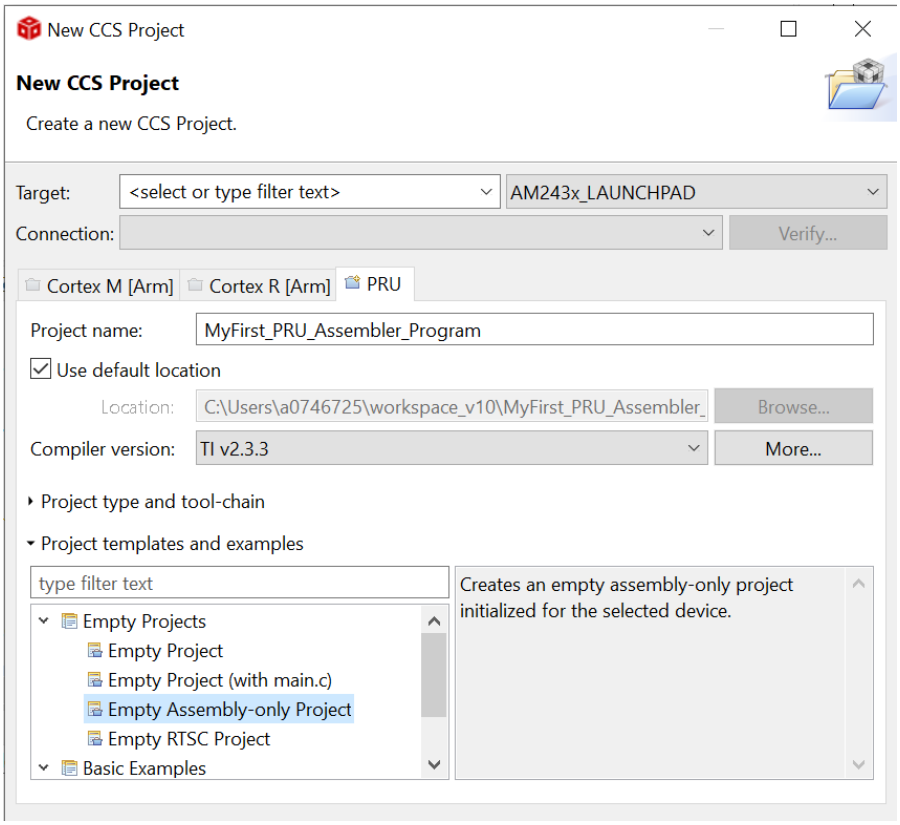


Figure 15 - New PRU assembler project

After Empty Assembly-only Project is created the Project Explorer view show only Includes and Debug folder. The linker command file can be added using Project Properties (right-mouse click on MyFirst_PRU_Assembler_Program Project, last option on pop-up menu). Figure 9 shows the settings for adding linker command file.

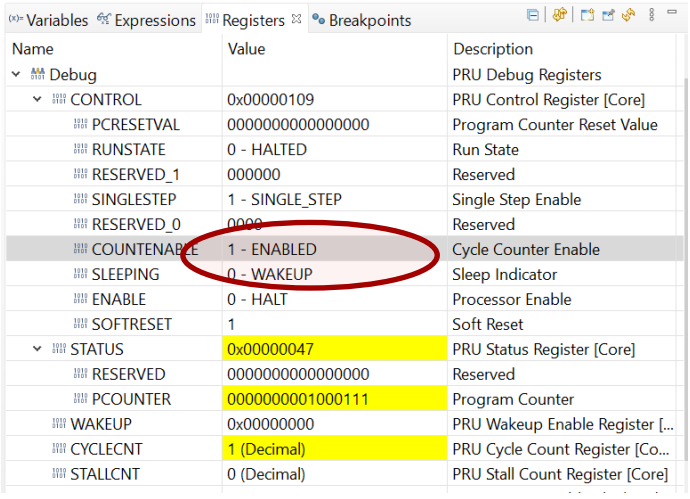


Figure 16- Enable PRU cycle counter

With source level debugger there are various options to modify program execution. A right click in the source code provides the options shown in figure 17. Besides breakpoints the user can use “Run to Line” or “Move to Line”. These functions are useful when testing instructions with different register settings. For example move to line of assembler instruction. Modify PRU register with different value and single step the instruction.

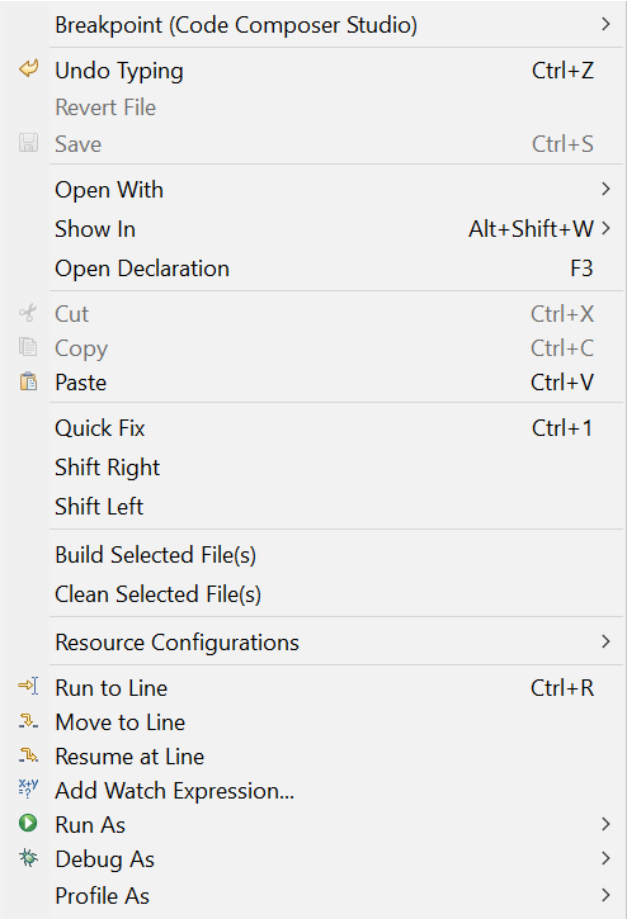


Figure 17- File options

5.3 *My first PRU program – mixed C with Assembler*

Learning goal:

- Example shows assembler routine called from C
- Example explains parameter transition from C to assembler
- Example explains return value from assembler to C
- Example discusses context safe in assembler when using conflicting registers with accelerators
    - Using Scratch Pad register

Projects in C language can be extended with assembler routines. The linker will handle integration of assembler functions with compiled C program. The example from appendix is a simple function call of assm_add with two arguments. The label is put between parallel bars "||" to indicate the function name which is called from C program. The assembler function is declared in C file with arguments and return value. As described in first example, PRU registers R14 and R15 are used to hand over arguments in function calls. The return address is in higher word of register r3. In source level debugger a single step execution (F5 key) automatically jumps from C source into assembler source and back.

Extract assembly file:

```
      .sect    ".text:assm_add"
      .clink
      .global ||assm_add||

||assm_add||:

      ; arg1 is in R14, arg2 is in R15
      ; the return value is stored in R14

      ; add arg1 and arg2. Store the sum in the return register
      ADD             R14, R14, R15

      ; return from function assm_add
      JMP             r3.w2
```

Extract C file:

```
/* Declaration of the external assembly function */
uint32_t assm_add(uint32_t arg1, uint32_t arg2);

/*
 * main.c
 */
void main(void)
{
    /* The compiler decides where to store x, y, and z */
    uint32_t x = 1;
    uint32_t y = 2;
    uint32_t z = 0;

    a = 1;
    b = 2;

    while(1) {
        /*
         * use the assembly function to add x and y, then
         * store the sum in z
         */
        z = assm_add(x, y);
```

This example does not modify any other registers in the assembler function and therefore does not require to save and restore register context using scratch pad. There are 3 extra register banks which can be used for

context storage or exchange with other PRU. Scratch pad registers are accessed in single cycle using XIN or XOUT instructions in assembler. There are also broadside intrinsics which can be called from C source. For example:

```
xout    BANK0_ID, &r0, 27*4     ; save R0-r26
```

saves 27 register into first scratch pad bank.

```
xin     BANK0_ID, &r0, 27*4     ; save R0-r26
```

Restores the register context back into PRU registers. There is also an option to apply shift operation when transferring register content to and from scratch pad.

6 ARM Driver

6.1 PRU2HEX

Learning goal:

- Explains how PRU firmware is generated for ARM download vs CCS out file download
- Explains step to copy header file from PRU project to ARM project.

PRU firmware can be loaded using the .out file generated by the project build process. This method is typically used during development and source level debug. For production and test, the PRU firmware is loaded from ARM side. Pre-defined SDK examples can also be configured with SYSCONFIG tool which generates code including the driver to manage PRU code download.

Here we describe how the build process in CCS to generate a header file which contains PRU opcode in a structure. To enable PRU HEX Utility go to the properties of the project and click on Build -> PRU Hex Utility and enable the utility as shown in figure 22.

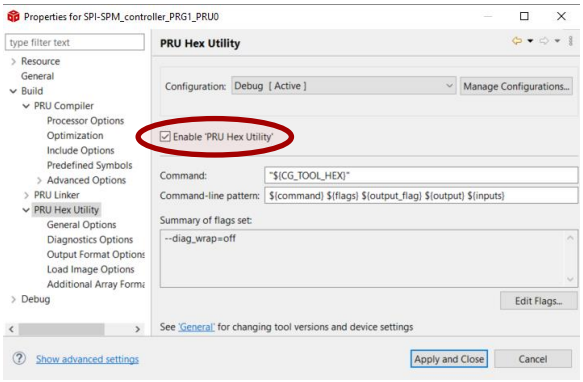


Figure 18 - Enable PRU Hex Utility

Next step is to specify the output file name which is found under General Options. Figure 23 highlights the entry for the header file.

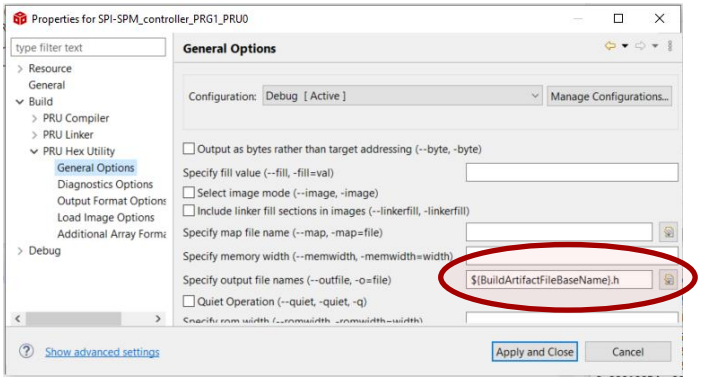


Figure 19 - Output file name

With Additional Array Format options a prefix for the output array name is provided.

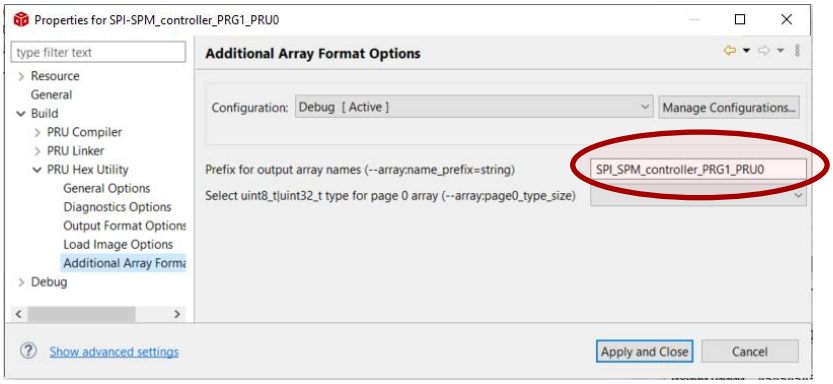


Figure 20 - Output name

Final step is to define the Output format to “Array (--array)”

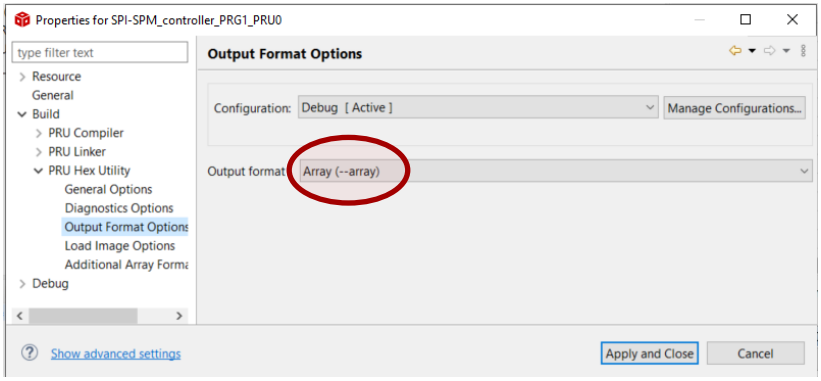


Figure 21 - Output format

Below is an example for generated header file which is used by PRU driver to load the firmware into PRU core.

```
const uint32_t lab_instr1_image_0[] = {
0x24ffffa2,
0x00422202,
0x2701e262,
0x09036202,
0x1d052222,
0x81031822,
0x91031842,
0x2ed00182,
0x240020ff,
0x10000000,
0x2a000000};
```

Figure 26 shows the Post-build step to copy the generated header file from PRU project to ARM project. The filename and folders need to be adjusted to the current project.

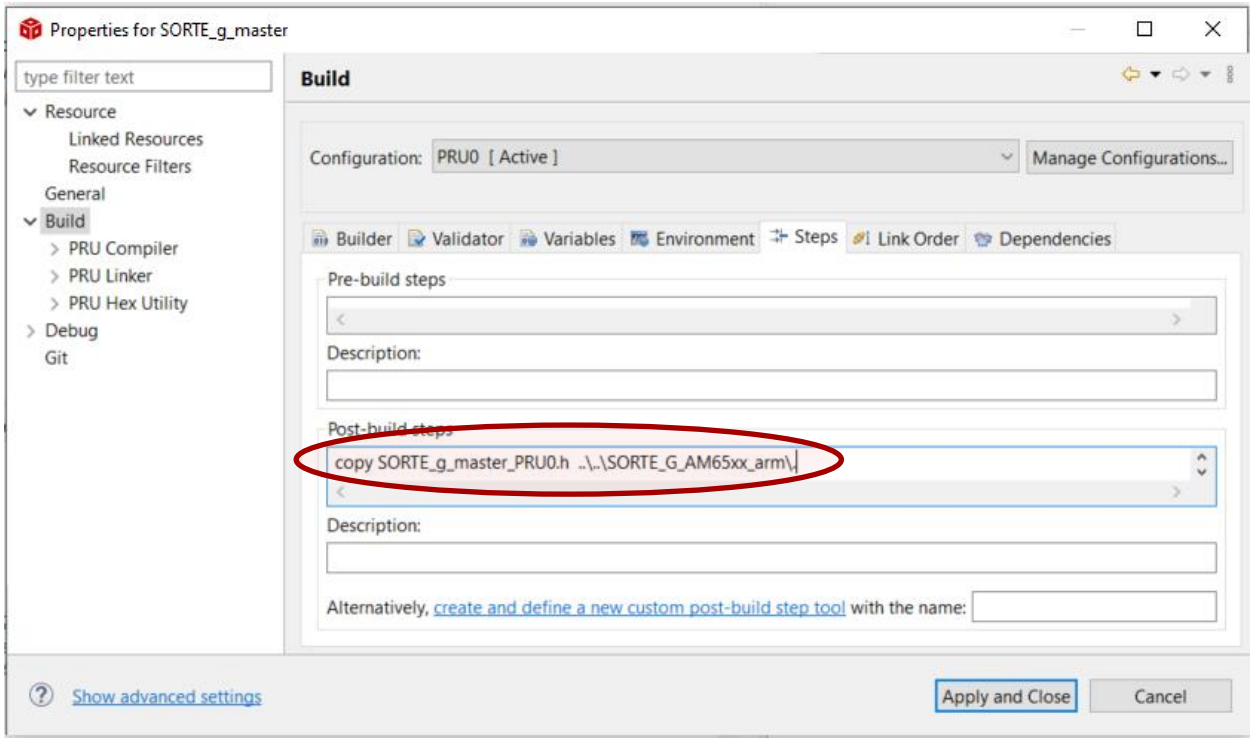


Figure 22 - Copy firmware file to ARM project

6.2 SYSCONFIG

Starting from empty project in MCU+ SDK, support for PRU (ICSS) can be added using SYSCONFIG tool which is integrated into CCS. Figure 27 shows basic PRU configuration with settings for PRU Core clock and Industrial Ethernet Peripheral (IEP) clock. The tool is integrated to CCS and opens with .syscfg file.

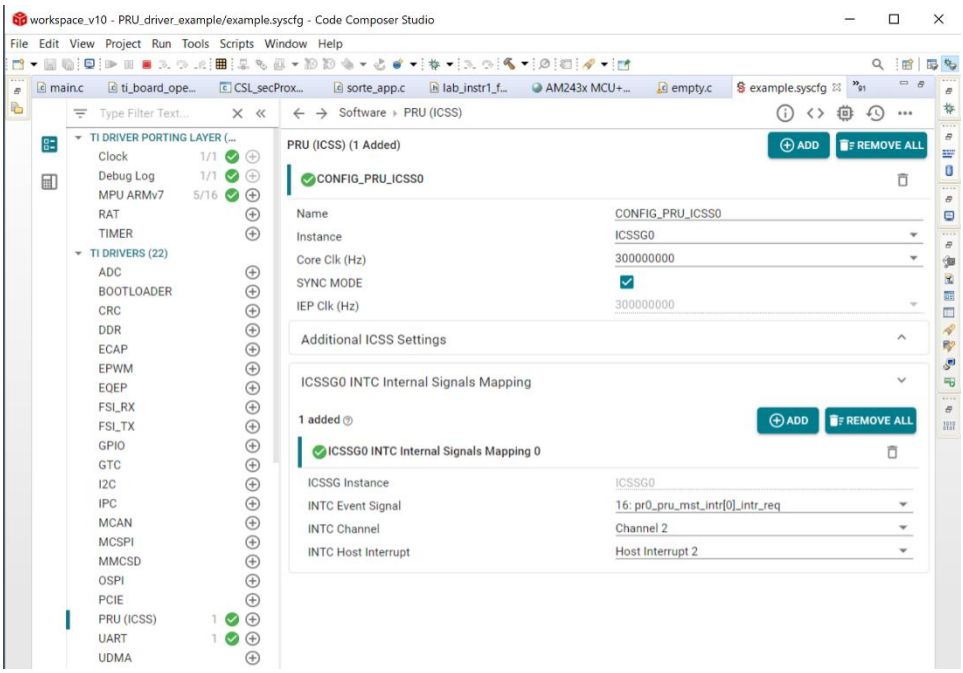


Figure 23 -SYSCONFIG PRU settings

6.3 PRU Driver API

Learning Goal:

- Explains basic steps to get handle, control PRU, download PRU and start PRU
- Discusses how to work with multiple PRUs from single ARM driver
- Discusses timing to change firmware on the fly
- Simple example (GPIO) which only connect to ARM in debugger

APIs for PRU_ICSS are documented in SDK+ online documentation:

https://software-dl.ti.com/mcu-plus-sdk/esd/AM64X/08_04_00_17/exports/docs/api_guide_am64x/group__DRV__PRUICSS__MODULE.html

Following steps are needed to configure, load and run PRU code from ARM side.

```
/** \brief Global Structure pointer holding PRUSS1 memory Map. */
PRUICSS_Handle gPruIcss0Handle;

void generic_pruss_init()
{
    HwiP_Params     hwiPrms;
    int32_t         retVal;
    uint32_t        intrNum;

    gPruIcss0Handle = PRUICSS_open(CONFIG_PRU_ICSS0);

    PRUICSS_disableCore(gPruIcss0Handle, PRUICSS_PRU0);

    /* clear ICSS0 PRU data RAM */
    gPru_dramx = (void *)((((PRUICSS_HwAttrs *)(gPruIcss0Handle->hwAttrs))->baseAddr)
+ PRUICSS_DATARAM(PRUICSS_PRU0));
    memset(gPru_dramx, 0, (4 * 1024));

    /*load firmware from structure pru0_image_0 generated by PRU Hex Utility */
    PRUICSS_writeMemory(gPruIcss0Handle, PRUICSS_IRAM_PRU(PRUICSS_PRU0),
                    0, (uint32_t *) pru0_image_0,
                    sizeof(pru0_image_0));

    PRUICSS_resetCore(gPruIcss0Handle, PRUICSS_PRU0);

    /*Run firmware*/
    PRUICSS_enableCore(gPruIcss0Handle, PRUICSS_PRU0);

}
```

The complete ARM C source file for the driver is in the Appendix. Same procedure needs to be followed to handle more PRUs the names are defined in the corresponding header file – drivers/pruicss.h . Below the defines for other PRU cores.

```
#define PRUICSS_PRU0                                    (0U)
```

```
#define PRUICSS_PRU1                    (1U)
#define PRUICSS_RTU_PRU0                (2U)
#define PRUICSS_RTU_PRU1                (3U)
#define PRUICSS_TX_PRU0                 (4U)
#define PRUICSS_TX_PRU1                 (5U)
```

The AM243x/AM64x has in total 12 PRU cores. Each core can be loaded with new firmware during ARM run-time. This allows for on-the fly change of PRU functions with new functions loaded and started in less than 1 ms. For bigger projects where PRU instruction memory is limited the firmware can load initialization code first followed by operational code and in case of errors diagnostic code.

6.4 Boot from Flash

- Explains all the steps from PRU build, ARM build, flash build, flashing the example and booting from flash.

(to be completed – refer to MCU+ SDK)

7 Appendix

7.1 References

PRU Optimizing C/C++ Compiler User's Guide

PRU Assembly Language Tools User's Guide

PRU assembly instructions

AM64x/AM243x Technical Reference Manual

AM243x Datasheet

7.2 PRU IO poster

PRU R30/31 Input/Output modes

- Each PRU processor implements fast GPO through R30 register and GPI through R31 register
  - PRU has full input and output control on all interfaces
  - RTU_PRU and TX_PRU see R31 input and can process receive in parallel

General-Purpose Input modes (R31)

Direct Input	**GPI [19:0]** feeds directly into the PRU R31 in Default state There are **80 general-purpose inputs** in total.
16-bit parallel capture	**DATAIN[0:15]** is captured by the pos_edge or neg_edge of **CLOCKIN R31[16]** programmable through the PRU_ICSSG CFG register → If clocking is configured to be **positive/negative**, then it will **equal PRUn_Clock/PRUn_Clock inverted**
28-bit Serial shift in mode	**DATAIN is sampled** and **shifted into a 28-bit shift register** on an internal clock pulse. The **shift rate** is controlled by the **effective divisor of two cascaded dividers applied to the ICSSG_CORE_CLK clock** (150MHz/ 200MHz/ 225MHz) → It also supports **Start Bit Detection (SB), Shift Counter (Cnt_16)** and **Stop/Freeze** current shift operation and disable search for new Start Bit
MII_RT	mii_rt_r31_status [29:0] internally driven by the MII_RT module Enabled by ICSSG_GPCFG0_REG[1-0] PRU0_GPI_MODE register (value: 3h), where n = 0 or 1
9x Sigma Delta	Integrator counts the number of 1's per clock event. **Each channel has three cascaded counters**, which are the accumulators for the **Sinc3 filter.** Each **counter and accumulator are 28 bits.** This sample counter **updates the count value** on the effective clock event for that channel. Each channel also **contains a programmable counter compare block.**

General-Purpose Output modes (R30)

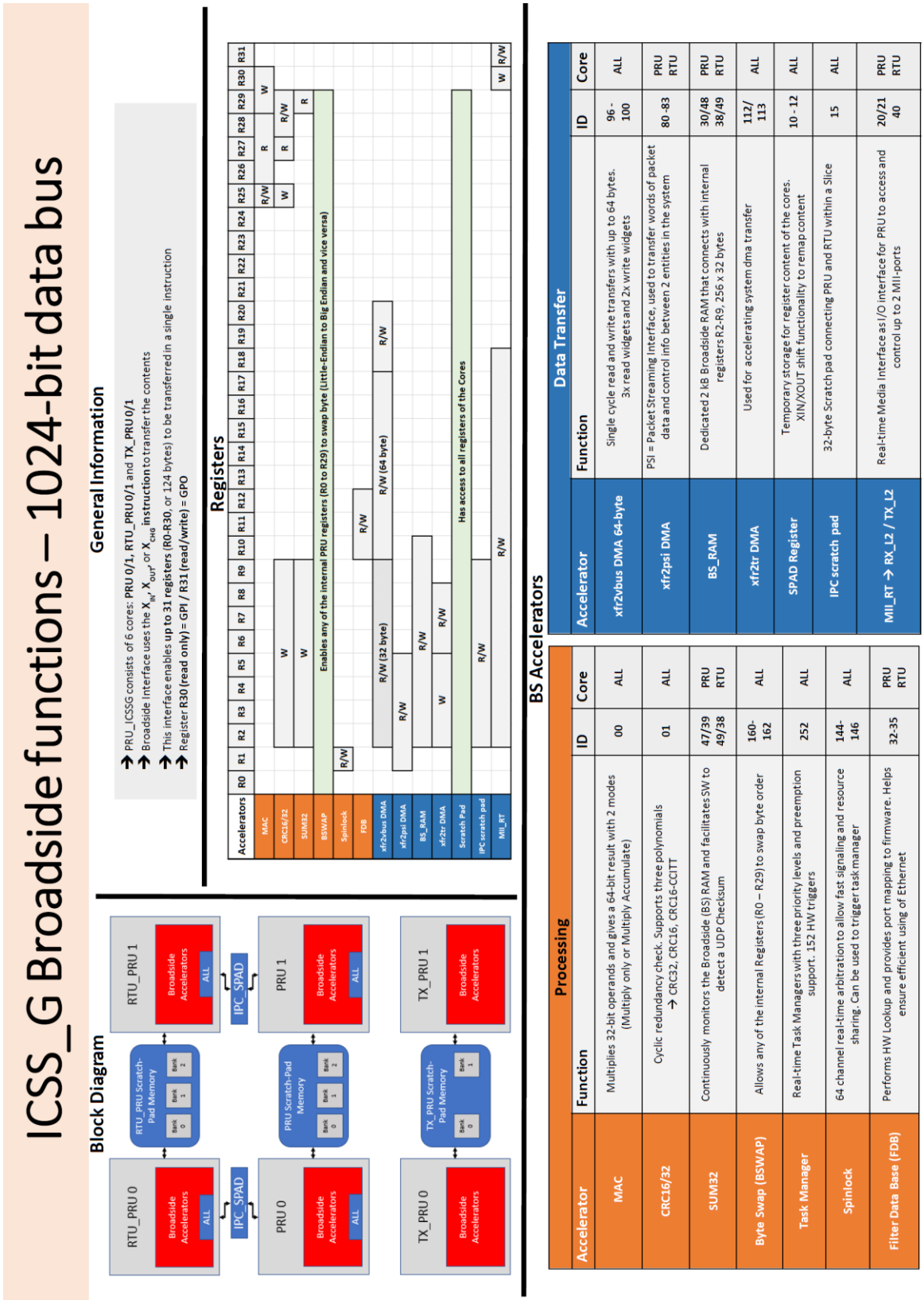
Direct Output	**PRU0_r30[0:19]** feeds directly into PRU0_GPO[0:19] There are **40 total general-purpose outputs**
Serial shift out mode	Data is **shifted out of PRU0_r30[0]** (PRU0_DATAOUT) on **every rising edge** of PRU0_r30[1] (PRU0_CLOCK) The **shift rate** is controlled by the **effective divisor of two cascaded dividers applied to the ICSSG_CORE_CLK clock** (150MHz/ 200MHz/ 225MHz) → Supports **2 modes**: Free Running Clock Mode (default) and Fixed Clock Count Mode

GPIO (R30/R31)

INTC (R31)	Interrupt controller (INTC) **maps interrupts** coming from different parts of the device to a reduced set of PRU_ICSSG interrupt channels, uses **bit 30, 31** Capturing **up to 160 Events,** supports **20 output interrupt channels,** each **event/host can be enabled/disabled**
3x Peripheral interface	**3 channels** with baud range from 100 kHz to 16 MHz, **The Peripheral Interface's I/Os** are **multiplexed** with the PRU GPI/GPO signals **TX FIFO** size of 32 bits, **RX FIFO** size of 4 bits

I/O	GPI/O0	GPI/O1	GPI/O2	GPI/O3	GPI/O4	GPI/O5	GPI/O6	GPI/O7	GPI/O8	GPI/O9	GPI/O10	GPI/O11	GPI/O12	GPI/O13	GPI/O14	GPI/O15	GPI/O16	GPI/O17	GPI/O18	GPI/O19
Direct Input	GPI0	GPI1	GPI2	GPI3	GPI4	GPI5	GPI6	GPI7	GPI8	GPI9	GPI10	GPI11	GPI12	GPI13	GPI14	GPI15	GPI16	GPI17	GPI18	GPI19
Parallel Capture	DATAIN0	DATAIN1	DATAIN2	DATAIN3	DATAIN4	DATAIN5	DATAIN6	DATAIN7	DATAIN8	DATAIN9	DATAIN10	DATAIN11	DATAIN12	DATAIN13	DATAIN14	DATAIN15	Clocking			
28-bit Shift In	DATAIN																			
9x Sigma Delta	SD0_CLK	SD0_D	SD1_CLK	SD1_D	SD2_CLK	SD2_D	SD3_CLK	SD3_D	SD4_CLK	SD4_D	SD5_CLK	SD5_D	SD6_CLK	SD6_D	SD7_CLK	SD7_D	SD8_CLK	SD8_D		
Direct Output	GPO0	GPO1	GPO2	GPO3	GPO4	GPO5	GPO6	GPO7	GPO8	GPO9	GPO10	GPO11	GPO12	GPO13	GPO14	GPO15	GPO16	GPO17	GPO18	GPO19
Serial shift out mode	DATAOUT	CLOCKOUT																		
3x Peripheral interface	PERIF0_CLK	PERIF0_Out	PERIF0_OUT_EN	PERIF1_Out	PERIF1_CLK	PERIF1_OUT_EN	PERIF2_CLK	PERIF2_Out	PERIF2_OUT_EN	PERIF0_IN	PERIF1_IN	PERIF2_IN								

7.3 PRU broadside poster



IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265