

Contents

6.	Virtual Prototype: ARMulator.....	6-1
6.1.	Overview.....	6-1
6.2.	Background information	6-1
6.2.1.	System synchronization	6-1
6.2.2.	ARMulator C hardware model	6-3
6.3.	Instructions	6-5
6.3.1.	RGB2YUV ARMulator hardware model.....	6-5
6.3.2.	Building hardware model using VC++'s NMAKE	6-7
6.3.3.	Building & debugging hardware model using VC++	6-8
6.4.	Exercise	6-13
6.5.	Reference	6-14
6.6.	Appendix.....	6-15

6. Virtual Prototype: ARMulator

6.1. Overview

Virtual prototype enables designer to verify the system early at a higher abstraction level during the development. Virtual prototype can be implemented using SystemC, SpecC, or other high level languages. In this lab, we implement virtual prototype with ARMulator. ARMulator supports adding user designed hardware model to simulate together with its ISS. This lab demonstrates adding an RGB2YUV hardware model into ARMulator. This high level hardware model is written in C using ARMulator's hardware model C constructs. You will learn the followings in this lab:

1. Be able to write and add C hardware model for ARMulator.
2. Learn how to write simple drivers for hardware.
3. Know system synchronization schemes: interrupt and polling

6.2. Background information

6.2.1. System synchronization

In order to let the IPs operate together with the system, synchronization between the ARM core and the IPs is needed. Synchronization lets the processor know when the assigned task of an IP is done, so the processor knows when and where to acquire the results of an IP's task. There are two basic synchronization schemes: interrupt and polling.

Interrupt:

An IP device signals an interrupt when it completes its tasks enabled by ARM core. We say that the IP "raised an interrupt request (IRQ)". This IRQ tells the ARM core that it has finished its task, and requests to be handled.

Figure 1 shows the basic concept when several IPs raise interrupts at the same time. The IRQs are sent to the interrupt controller. Then interrupt controller will check if the IRQs are enabled; if so, it updates the IRQ status registers within itself to indicate which devices requested IRQs. The sorted out IRQ is sent to the ARM core to request the handling for the corresponding IP. The ARM core will stop the current task at a proper time, determine which IRQ should handled first according to programmed priority, and then backup the current CPSR content to the SPSR, so it can resume the original task after finishing handling the IRQ. If it is a nested IRQs, the content of the SPSR would be pushed into the stack. The ARM core will issue the corresponding interrupt service routine (ISR) by checking the interrupt status registers in the

interrupt controller to determine which device requested the interrupt. The ISR shall perform the corresponding operations and then clear the interrupt request by writing a signal to clear interrupt signal of the corresponding IP, which is IP0 in this case. After the ISR is done, ARM core would pop the CPSR's original content and restore other related context to continue its original task.

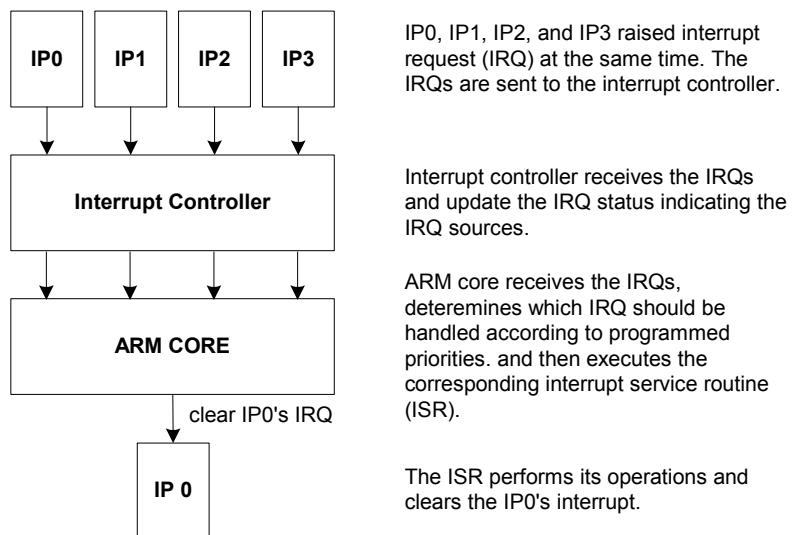


Figure 1. A simple flow of how interrupt works.

Interrupt enables ARM core to continue working while the IP device has been enabled. Yet the IRQ might be some how a little unpredictable since you don't know when the IP device would raise IRQ. Once an IRQ is raised, the core must handle it with its ISR, this would interrupt the core's original task. And using interrupt requires an interrupt controller hardware unless there's only one device that would raise interrupt, which is often not the case.

Polling:

The ARM core keeps accessing a certain register in the IP which indicates whether it has completed its task enabled by the ARM core for a certain time interval. Once the IP has done its task, the register changes its value, so the ARM core could know the IP is ready and the IP requires to be handled when ARM core accesses the register. The action of continuous accessing and checking the register with a certain time interval is called "polling".

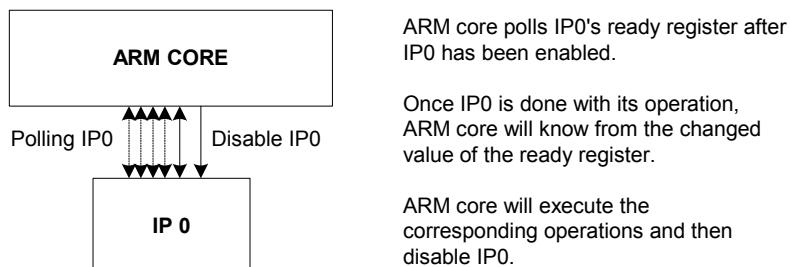


Figure 2. A simple flow of how polling works.

Figure 2 illustrates the basic way polling is carried out. The ARM core enables IP0 and then keeps “polling” it. The ARM core checks the ready register of IP0 to see if it has finished its task. After IP0 has done its task, the value of the ready register would change to indicate it has finished its task. When ARM core polls, it will know IP0 is done and can start the corresponding operations after the completion of IP0’s task. The ARM core will finish the corresponding operations and disable IP0. Once IP0 is disabled, the ready register would change to its original not-ready value.

Note that it's often not possible for ARM core to perform parallel operation with the IP using polling since the core has to keep polling the IP device. And it is up to the software program to determine when to handle the ready IP device. The advantage of using polling is that no additional hardware is required compared to using interrupt synchronization scheme.

6.2.2. ARMulator C hardware model

ARMulator supports IP designers to design their IP’s hardware model to run simulations together. ARMulator is a cycle-accurate emulator, it consists of an ARM core instruction-set simulator (ISS) program and several other peripheral modules. New device can be designed with C/C++ language using certain predefined functions interface for hardware modeling with ARMulator. The design is built as dynamic link libraries (DLLs). By adding IP’s hardware model’s DLL and modifying corresponding component description parameters, the hardware model can be included into the simulation.

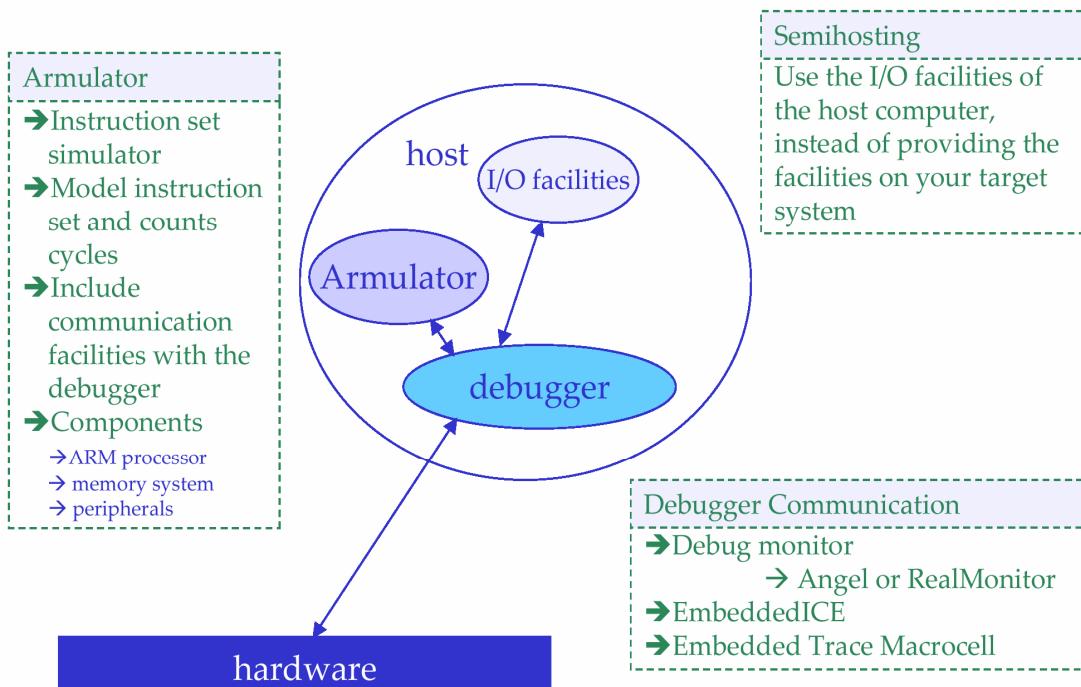


Figure 3 Overview of ARMulator

An ARMulator hardware model must include ARMulator basic model interface. A basic model interface includes three parts:

1. Data structure declaration
2. Initialization
3. Finalization

Data Structure Declaration:

Data structure is declared using BEGIN_STATE_DECL() and END_STATE_DECL() macros. Private data structure is declared as follow.

```
BEGIN_STATE_DECL(YourModel)
/*
 * your private data here
 */
END_STATE_DECL(YourModel)
```

These macros declare a data structure:

```
typedef struct YourModelState
```

This data structure includes your private data you put between the macros and the following predefined data fields:

```
Toolconf config
Const struct RDI_HostInterface *hostif
RDI_ModuleDesc coredesc
RDI_ModuleDesc agentdesc
```

Initialization

We use BEGIN_INIT() and END_INIT() macros to delineate the initialization functions of the model. In the initialization function, your model must initialize any private variable and install any callback. Two variables are provided in the initialization:

→ **Bool** coldboot
TRUE if ARMulator is initializing, FALSE if a new image is being downloaded from the debugger.

→ **YourModelState** *state

A pointer points to the private state data structure. Memory for this is allocated and declared by the initialization macro, and the predefined data fields are initialized.

Finalization

We use BEGIN_EXIT() and END_EXIT() macros delineate the finalization function for the model. The finalization function is called when ARMulator is closing down. The following local variable is provided in the finalization function:

```
YourModelState *state
```

Your model must un-install any callbacks in the finalization function. The END_EXIT() macro frees memory allocated for state.

There are several useful ARMulator functions; you can refer to *ARM Debug Target Guide* for further details on ARMulator hardware modeling.

6.3. Instructions

6.3.1. RGB2YUV ARMulator hardware model

The hardware model taken for example in this lab performs RGB2YUV operation. In image processing, red, green, and blue signals are taken in from CCD sensors. However, image encoder such as JPEG encoder adopts luminance (Y) and two chrominance components (U, V) to represent image. Therefore a transformation from RGB to YUV is necessary and is often the first step taken in a JPEG encoder.

The formula to transform RGB to YUV is shown in the following equation:

Function:

$$\begin{bmatrix} Y \\ C_B \\ C_R \end{bmatrix} = \begin{bmatrix} 0.257 & 0.504 & 0.098 \\ -0.148 & -0.291 & 0.439 \\ 0.439 & -0.368 & -0.071 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix}$$

Table 1 Memory map of RGB2YUV

Address	Read access	Write access
RGB2YUV_base	RGB2YUV Control register [31:2]: Reserved [1]: done bit 0: computation NOT done 1: computation done [0]: enable status register 0: disabled 1: enabled	RGB2YUV Control register [31:3]: Reserved [2]: clear interrupt bit 0: do nothing 1: clear interrupt [1]: Reserved [0]: enable register 0: disable 1: enable
RGB2YUV_base + 0x04	Red input register [7:0]: Red value	Red input register [7:0]: Red value
RGB2YUV_base + 0x08	Green input register [7:0]: Green value	Green input register [7:0]: Green value
RGB2YUV_base + 0x0C	Blue input register [7:0]: Blue value	Blue input register [7:0]: Blue value
RGB2YUV_base + 0x10	Y output register [7:0]: Y value	Reserved
RGB2YUV_base + 0x14	U output register [7:0]: U value	Reserved
RGB2YUV_base + 0x18	V output register [7:0]: V value	Reserved

The transform is a 3x3 matrix multiplication and addition. The hardware is assumed to have 3 multipliers and 3 adders, hence 3 cycles are needed to complete a transformation from RGB to YUV. The memory map for this device is listed in Table 1. The based address is set at 0xC2100000 and each register is word aligned.

The procedure of using RGB2YUV is illustrated in Figure 4. A master (ARM or DMA) first load the RGB values by writing Red, Green, and Blue register at offset 0x04, 0x08, and 0x0C. After loaded the data to be processed, the master writes 0x01 to Control register at RGB2YUV base address, which is 0xC2100000. By writing 1 to bit 0 (enable register), RGB2YUV starts computing. Upon completion, the interrupt signal would be set and done bit in Control register would be set also. If synchronization is carried out using interrupt, corresponding ISR for this interrupt will clear the interrupt and disable the device, some may also retrieve the data to a target position. The function call graph of RGB2YUV is illustrated in Figure 5.

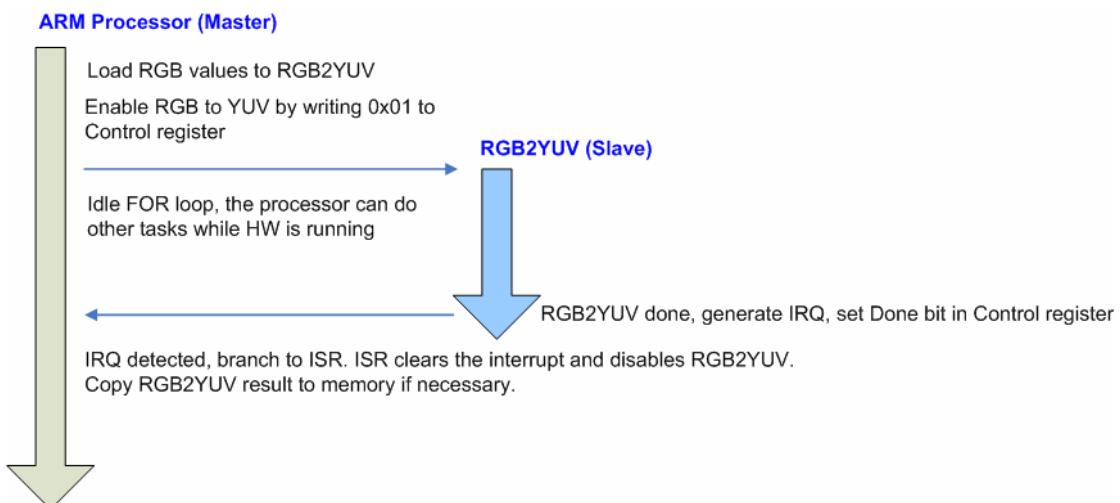


Figure 4 Procedure of using RGB2YUV

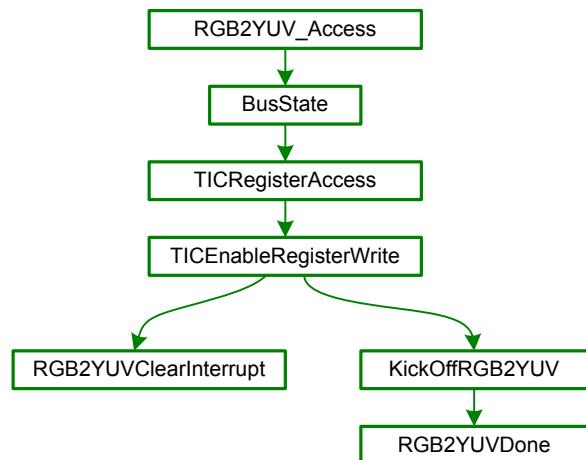


Figure 5 Function call graph in RGB2YUV C hardware model

6.3.2. Building hardware model using VC++'s NMAKE

1. Search for vcvar32.bat using WinXP's file search utility.
2. In WinXP, **Program Files** → **Accessories** → **Command Prompt**, a **command prompt** window appears.
3. Execute vcvar32 in the command prompt. This sets the environment for Microsoft Visual C++ NMAKE utility.
4. Copy **RGB2YUV.c** into **\$ARM_INSTALL_DIR\Bin**, **\$ARM_INSTALL_DIR** is the install directory of ARM ADS. For PCs in ED414, **\$ARM_INSTALL_DIR** is **C:\Program Files\ARM\ADSv1_2**.
5. Copy directory **RGB2YUV.b** into **\$ARM_INSTALL_DIR\ARMulator\armulext**. There is an **intelrel** directory within this directory, a **Makefile** is provided in **intelrel**.
6. Change current directory in **command prompt** to **\$ARM_INSTALL_DIR\ARMulator\armulext\RGB2YUV.b\intelrel**.
7. Type **nmake** in **command prompt** to make the hardware model dynamic link library. The dynamic link library file is **RGB2YUV.dll**, it is located in **intelrel** after **nmake** is completed.
8. Copy **RGB2YUV.dll** from **intelrel** to **\$ARM_INSTALL_DIR\Bin**
9. Create **RGB2YUV.dsc** in **\$ARM_INSTALL_DIR\Bin** with the following content.

```
;; ARMulator configuration file type 3
{
    Peripherals
    {
        RGB2YUV
        MODEL_DLL_FILENAME=RGB2YUV
    }
    {
        No_RGB2YUV=Nothing
    }
}
```

10. Modify **default.ami** in **\$ARM_INSTALL_DIR\Bin**, add RGB2YUV part below timer part.

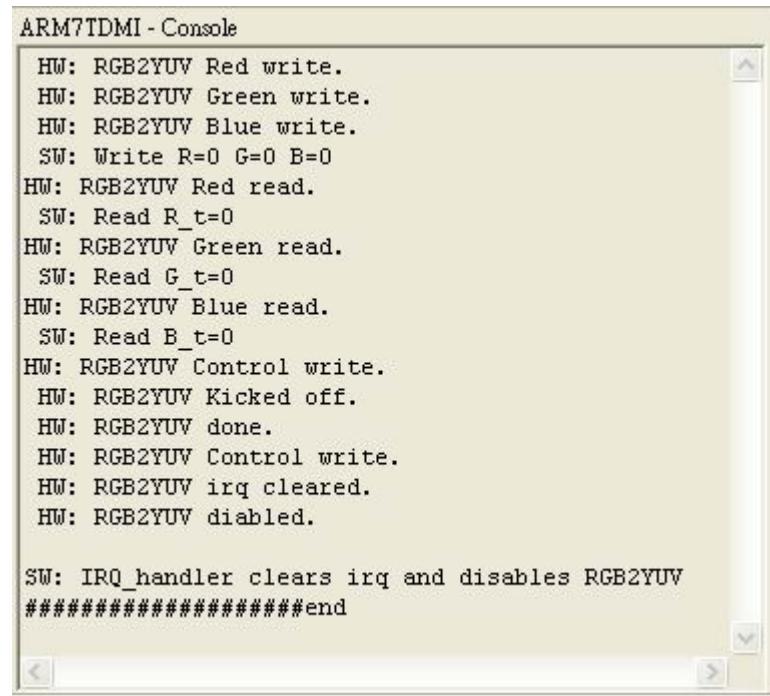
```
{ Timer=Default_Timer
}

{ RGB2YUV=Default_RGB2YUV
}
```

11. Modify **peripherals.ami** in **\$ARM_INSTALL_DIR\Bin**, add RGB2YUV part below timer part.

```
{ Default_RGB2YUV=RGB2YUV
Range:Base=0xC2100000
}
```

12. Open **RGB2YUV_demo.mcp** project file with CodeWarrior. This project tests RGB2YUV hardware.
13. Make the project and run. The execution results should be the same as follow.



The screenshot shows a Windows-style console window titled "ARM7TDMI - Console". The window contains a scrollable text area displaying a log of system events. The log includes various hardware (HW) and software (SW) interactions related to RGB2YUV operations, such as writes to Red, Green, and Blue channels, reads from those channels, control writes, and IRQ handling. The log concludes with a message indicating the IRQ handler has cleared the IRQ and disabled the RGB2YUV module.

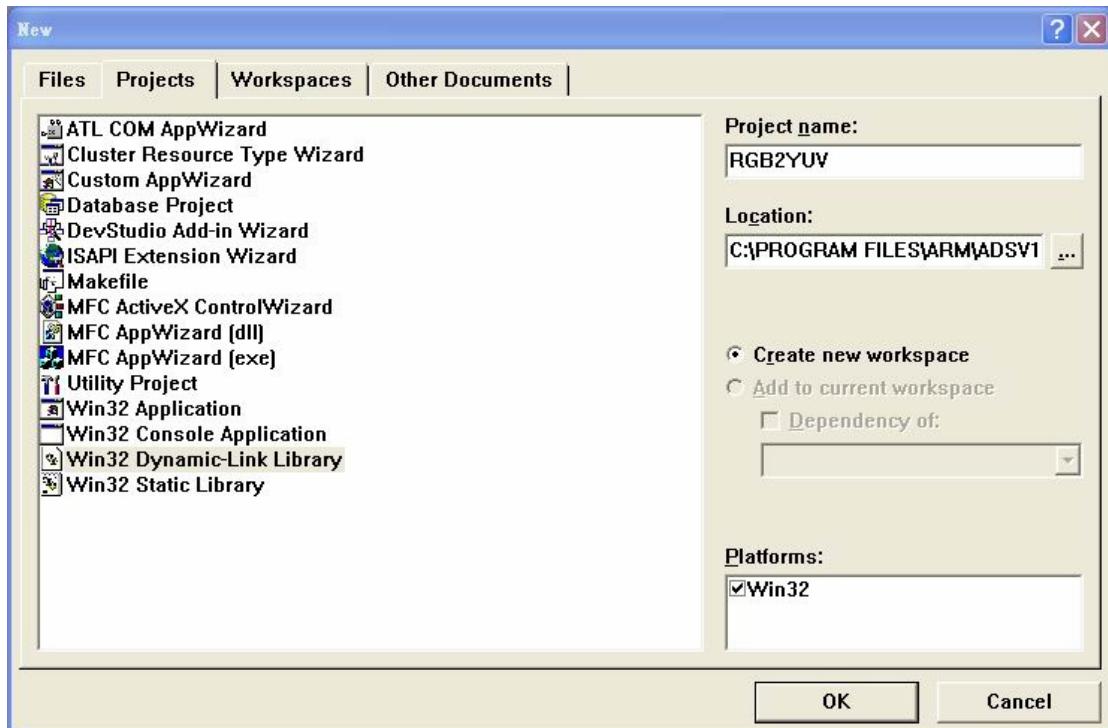
```
ARM7TDMI - Console
HW: RGB2YUV Red write.
HW: RGB2YUV Green write.
HW: RGB2YUV Blue write.
SW: Write R=0 G=0 B=0
HW: RGB2YUV Red read.
SW: Read R_t=0
HW: RGB2YUV Green read.
SW: Read G_t=0
HW: RGB2YUV Blue read.
SW: Read B_t=0
HW: RGB2YUV Control write.
HW: RGB2YUV Kicked off.
HW: RGB2YUV done.
HW: RGB2YUV Control write.
HW: RGB2YUV irq cleared.
HW: RGB2YUV disabled.

SW: IRQ_handler clears irq and disables RGB2YUV
#####end
```

6.3.3. Building & debugging hardware model using VC++

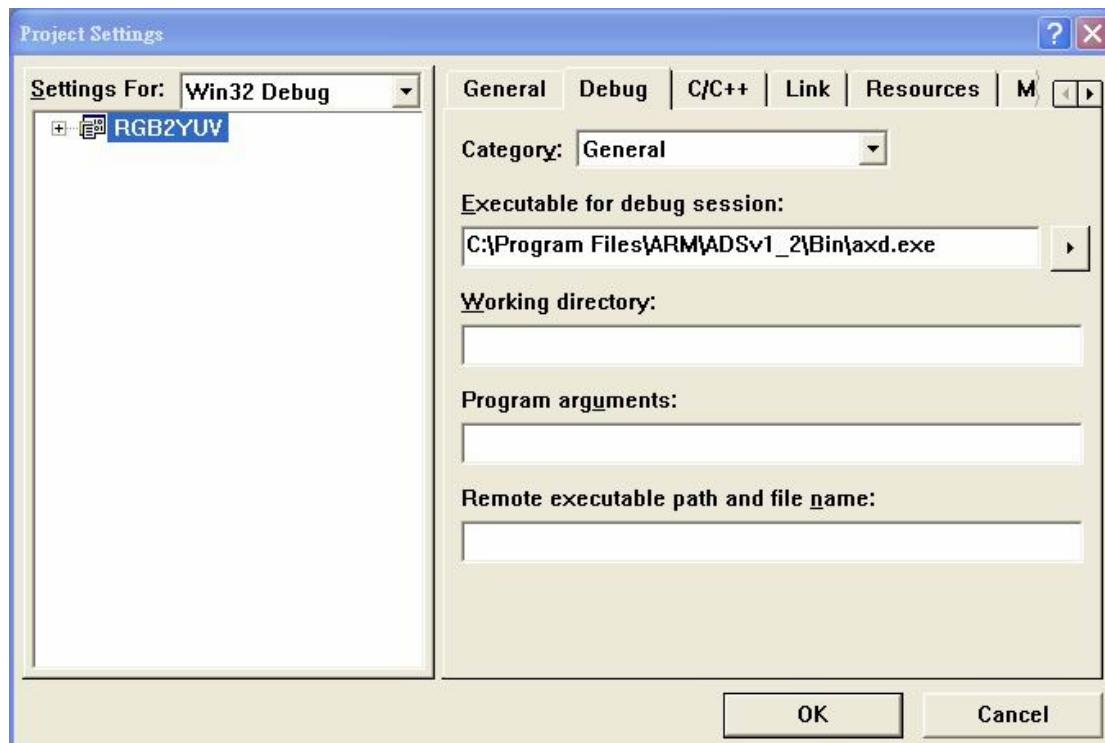
1. Copy **RGB2YUV.c** into **\$ARM_INSTALL_DIR\ARMulate\armulext**. Skip this step if you've already done so.
2. Copy **RGB2YUV.b** directory into **\$ARM_INSTALL_DIR\ARMulate\armulext**. Skip this step if you've already done so.
3. Make sure **RGB2YUV.dsc** is added, also make sure **default.ami** and **peripherals.ami** are modified as shown in **step 10** to **step 11** in **6.3.1**.
4. Start **Microsoft Visual C++ 6.0**
5. **File → New** to create a new empty project with the following settings. Press OK when you are done.

Project Type: Win32 Dynamic-Link Library
Project Name: RGB2YUV
Location: C:\Program Files\ARM\ADSv1_2\ARMulate\armulext\RGB2YUV.b

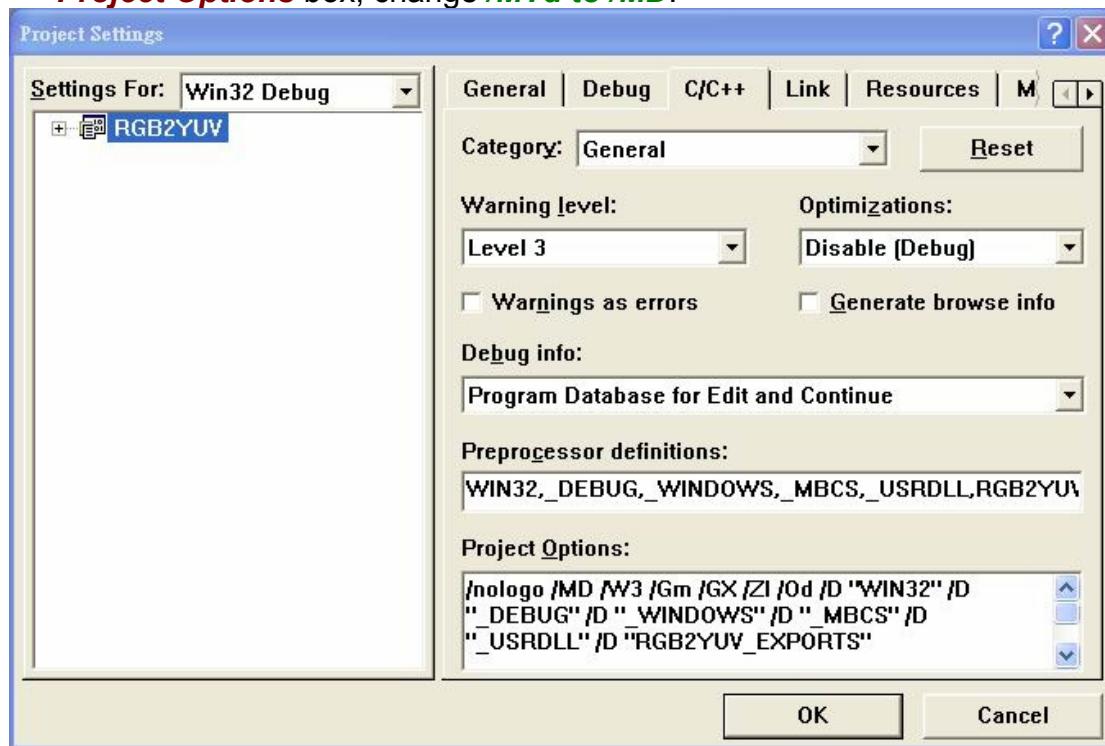


6. ***Project→Add To Project→Files*** to add following files into ***\$ARM_INSTALL_DIR\ARMulate\arm\ext*** to the project
RGB2YUV.c
sordi.def
version.rc
7. ***Project→Settings...*** or press ***ALT+F7*** to open project settings window.
8. Make sure the drop down list at the top-left displays Win32 Debug. If not, select this option.
9. Click ***Debug*** tab, choose category "***General***" from the drop-down menu. Type ***\$ARM_INSTALL_DIR\bin\axd.exe*** in ***Executable for debug session*** field.

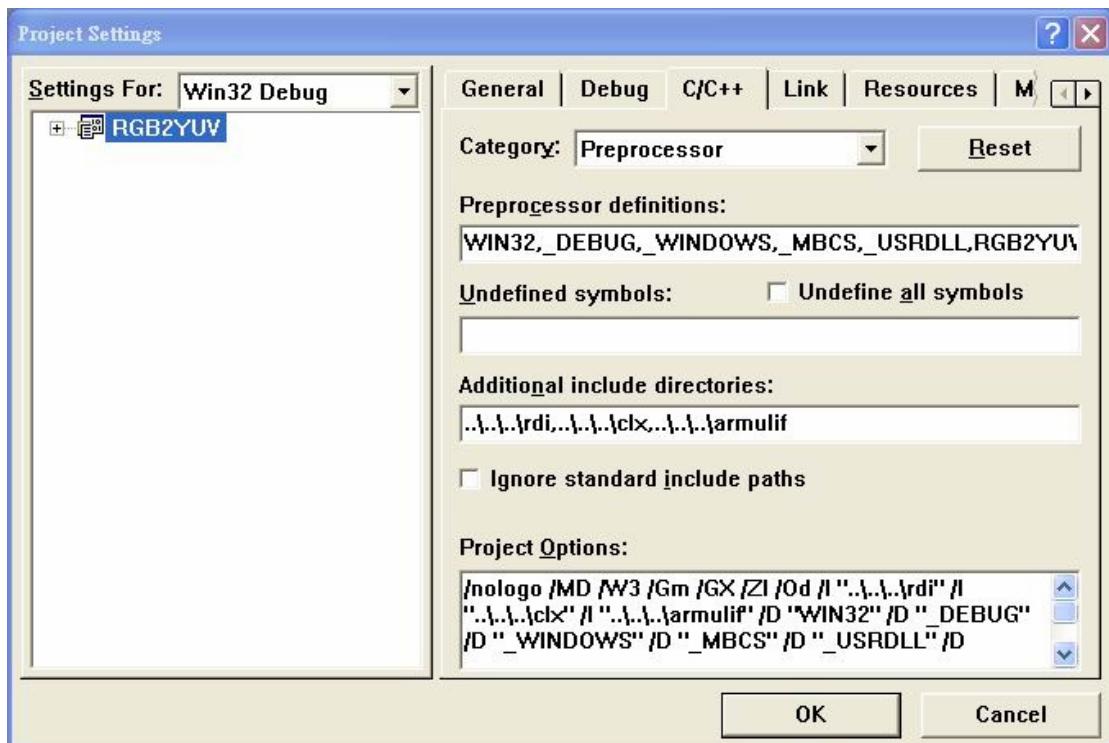
Virtual Prototype: ARMulator



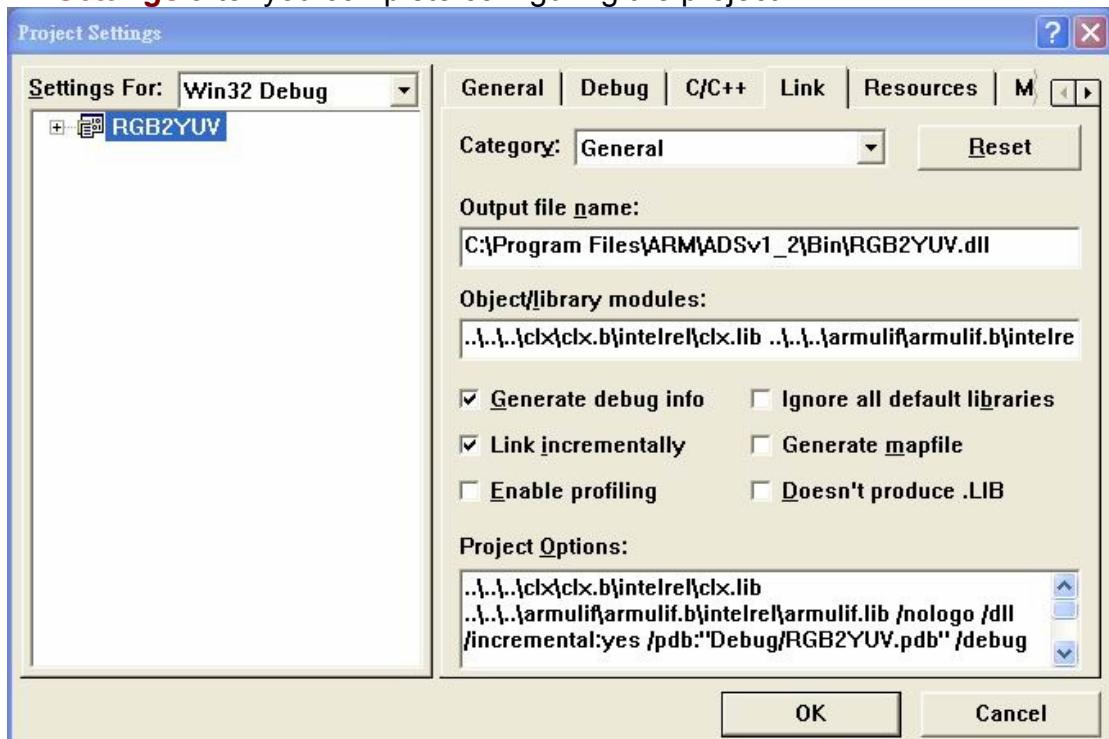
10. Click **C/C++** tab, choose category "**General**" from the drop-down menu. In **Project Options** box, change **/MTd** to **/MD**.



11. In **C/C++** tab, choose category "**Preprocessor**" from the drop-down menu. Specify **...\\..\\rdi**, **...\\..\\clx**, **...\\..\\armulif** in **Additional include directories** field



12. Click **Link** tab, choose category "**General**" and replace the contents of **Object/library modules** field with: `..\..\..\clx\clx.b\intelrel\clx.lib ..\..\..\armulif\armulif.b\intelrel\armulif.lib`. Type `$ARM_INSTALL_DIR\Bin\RGB2YUV.dll`. Close **Project Settings** after you complete configuring the project.



13. Build the project by **Build→Build RGB2YUV** or by pressing **F7**. This generates **RGB2YUV.dll** at **\$ARM_INSTALL_DIR\Bin**.

Virtual Prototype: ARMulator

14. Open **RGB2YUV.c** source code file and **set break points** at following lines: **211, 226, 248, and 306** by pressing **F9** at the lines.

```
static void RGB2YUUDone(void *handle)
{
    RGB2YUUState *ts = (RGB2YUUState*)handle;
    rgb2yuv_state *mysehw = &ts->RGB2YUU;
    Hostif_ConsolePrint(ts->hostif," HW: RGB2YUU done.\n");
    ARMulif_SetSignal( &(ts->coredesc), RDIPropID_ARMSignal_IRQ, Signal_On );
}

/*
Function Name: RGB2YUUClearInterrupt
Parameters: RGB2YUUState *ts
Return: void
Description: Clears the interrupt source
*/
static void RGB2YUUClearInterrupt(RGB2YUUState *ts)
{
    ARMulif_SetSignal( &(ts->coredesc), RDIPropID_ARMSignal_IRQ, Signal_Off );
    Hostif_ConsolePrint(ts->hostif," HW: RGB2YUU irq cleared.\n");
}

/*
Function Name: TICEnableRegisterWrite
Parameters: RGB2YUUState *ts,
           rgb2yuv_state *myRGB2YUU,
           ARMword *word   pointer to data to use for write.
Return: int - always return 0
Description: Breakout function for TICRegisterAccess - the final address decoder - to
            keep the complexity of one function down a little to keep it readable - this
            is hand generated not auto-generated code.
            This function performs the processing for a write to the RGB2YUU control
            registers. It updates the mirror struct members such as enabled/prescale.
            It processes changes to enable and prescale.
*/
static int TICEnableRegisterWrite(RGB2YUUState *ts, rgb2yuv_state *myRGB2YUU, ARMword *word)
{
    /* Update the internal representation first */
    myRGB2YUU->RGB2YUUEnableChangeMask = (getBit(myRGB2YUU->RGB2YUU_Control,0))^(getBit(*word,0));
    myRGB2YUU->RGB2YUU_Control = *word&0x00000005; // Only the least 3 bits are significant : Clear, Done,
    // Only bit 2 and 0 can be written

    then the access is passed on to the next level of memory by switch.
*/
static int TICRegisterAccess(RGB2YUUState *ts,ARMword addr,ARMword *word, unsigned /*ARMul_acc*/ acc)
{
    unsigned long maskedAddress = addr & 0x0000003F;
#ifdef VERBOSE
    printf("TICRegisterAccess: A:%08lx D:%08lx.\n", (unsigned long)addr,
        (word==NULL)?0UL:(unsigned long)*word);
#endif

    /* Yes it is a memory access - now is it a read or a write? */
    if ( ACCESS_IS_READ(acc) )
    {
        /* Read from registers */
        switch ( maskedAddress )
        {
            case 0x00:      //RGB2YUU Control
                // Only the least 3 bits are significant : Clear, Done, Enable
                *word = ts->RGB2YUU.RGB2YUU_Control & 0x00000007;
                break;
            case 0x04:      // read red
                //Hostif_ConsolePrint(ts->hostif,"read ");
                Hostif_ConsolePrint(ts->hostif," HW: RGB2YUU Red read.\n");
                *word = ts->RGB2YUU.RGB2YUU_Red;
                break;
        }
    }
}
```

15. Press **F5** to in VC++ 6.0 to launch AXD and ARMulator. Load **RGB2YUV_demo.axf** in AXD. Run the AXD image, when RGB2YUV hardware model is triggered, the process will halt at the break points set in **step 14**.

16. Observe how the hardware model works. By using VC++, you can debug much easier and faster. On occasion without VC++, you'll have to dump out internal information using **Hostif_ConsolePrint** routine

6.4. Exercise

Add a new hardware (*Matrix Transpose*, *MT*) model. This hardware transposes a 2x2 matrix. The input row data are changed into output column data ($\mathbf{A}=\mathbf{B}^T$). Each data is to be 8-bit data. This operation is assumed to take **2** cycles. The base address is **0xC2110000**. Name your module as **MT** and the name of the source code as **MT.c**.

Then write a simple test program to test your hardware model.

Please explain your **memory map** and source code to TA while demo.

Refer to *ARM Debug Target Guide* for ARMutator function definitions.

Hint: you can first replace **RGB2YUV** with **MT**, replace **rgb2yuv** with **mt**.

6.5. Reference

1. ARM Application Note 32: The ARMulator [DAI0032E]
2. ARM Debug Target Guide [DUI0058D]

6.6. Appendix

There are four most common functions we use in this lab. They are `ARMulif_Time()`, `ARMulif_ScheduleUntractable()`, `ARMulif_SetSignal()` and `Hostif_ConsolePrint()`.

ARMulif_Time:

This function returns the number of memory cycles executed since system reset. The syntax of this function is:

```
ARMTIME ARMulif_Time (RDI_ModuleDesc *mdesc)
```

`mdesc` is the handle for the core.

ARMulif_ScheduleUntractable :

This function is used to schedule a function to be executed after certain cycles. The syntax of this function is:

```
void ARMulif_ScheduleUntractable(RDI_ModuleDesc *mdesc,
                                  ARMuL_TimedCallBackProc *func,
                                  void *handle, ARMTIME when,
                                  ARMTIME period)
```

`func` is the function to be executed. `when` is the time for the function to be executed.

ARMulif_SetSignal:

This function is used to set the state of signals or properties. The syntax of this function is:

```
void ARMulif_SetSignal(RDI_ModuleDesc *mdesc,
                       ARMSignalType sigType, SignalState
                       sigState)
```

`sigType` is the signal to be set. `sigState` is either on or off.

Hostif_ConsolePrint:

This function is used to print some variables in ARMulator via RDI to the console window. The syntax of this function is:

```
void Hostif_Consoleprint(const struct RDI_HostosInterface
                         *hostif, const char *format, ... )
```

Virtual Prototype: ARMutator

hostif is the handle for the host interface. format is a pointer to a printf-style formatted output string. ... are a variable number of parameters associated with format. For example:

```
Hostif_ConsolePrint(state->hostif, "result=%d \n", result);
```