anadigm

App Note - 207

Application Note: State-Driven Control of a dpASP using a Microchip PIC.

Rev: 1.0.1

Date: 8th July 2009

This application note contains a total 3 files, if you have only this pdf text document, go here
http://www.anadigm.com/sup_AppNoteLib.asp
to find the source code and AnadigmDesigner2 document

TABLE OF CONTENTS

1 Purpose

When using low end microcontrollers (uC) to configure Anadigm's range of dpASP devices, it may not be possible to use the algorithmic code generated by AnadigmDesigner®2 (AD2). This is because the performance of the uC may be too slow due to insufficient memory and computing capability. In this case it is necessary for the user to write his own state-driven code.

The purpose of this document is to describe how to create the code for a simple example of an embedded system consisting of a PIC microcontroller controlling and an Anadigm AN221E04 dpASP. The uC code described in this document is intended to be very efficient in terms of both memory and speed, and is therefore suitable for even the simplest of microcontrollers.

The points covered in this document include:

- Creating the primary configuration data
- Writing the code to execute a primary configuration on power up
- Creating the reconfiguration data
- Writing the state driven reconfiguration code

2 Writing the Primary Configuration Code

2.1 Create the Primary Circuit

The first thing to be done is to create the primary circuit in AnadigmDesigner2, an example is shown in figure 1.

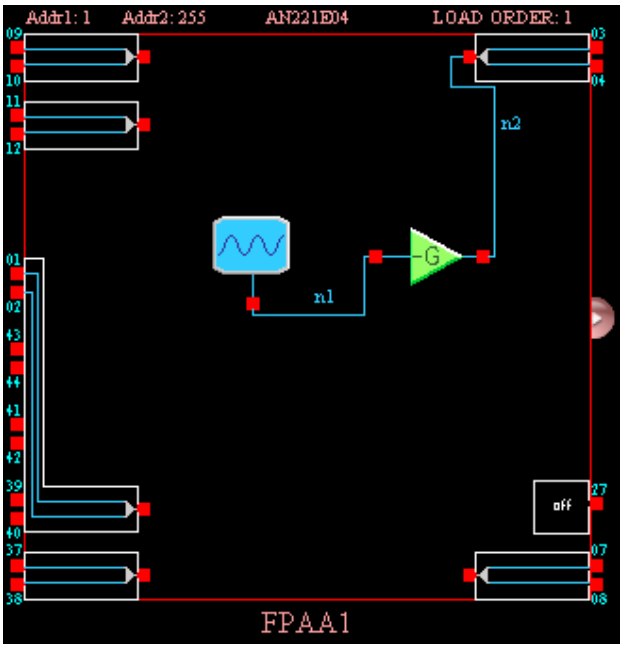

Figure 1

This circuit consists of an oscillator CAM whose frequency we wish to control, and a GainInv CAM. Save this circuit to the required directory.

2.2 Create the Primary Configuration Data

Go to the “Dynamic Config.” menu within AnandigmDesigner2 (AD2) and select “State-driven Method…”.

Under the “States” tab click on “Complete Chips…”, select FPAA1 (or whatever the relevant chip is called) and click “OK”.

Select the “Transitions” tab and check the box next to FPAA1 to indicate that this is a primary state.

Select “Generation” tab, click on the radio button labelled “Generate C Formatted Configuration Text Files”. Also click on “All transitions in one file”. Make sure that the destination directory is pointing to the required directory.

Now click on the “Generate” button. A warning will come up saying that primary configurations will be generated but transitions will not. Click on “OK”.

A text file will be generated that contains the primary configuration data.

Close the “State-driven Dynamic Configuration” window

2.3 Write the Primary Configuration Code

Note that when using more complex microprocessors or a PC, one would not normally copy and paste the array from text files in this way but generate state-driven C-code files and include these AD2 generated files in the project, then reference the arrays and functions in these C-code files.

We are avoiding a possible problem with low end uC’s. Some of the functions in the state-driven code generated by AD2 are complex and may not be supported or compatible with simple devices. This is why we are writing our own code below:

The code generated in 2.2 above should be copied and pasted into the C-code in the form of an array as shown below.

```
#define Primary_Config_Size          169

const an_Byte Primary_Data[] =
{
      0x00, 0x00, 0x00, 0x00, 0x00, 0xD5, 0xB7, 0x22,
      0x00, 0x80, 0x01, 0x05, 0xCC, 0x00, 0x0C, 0x20,
      0x00, 0x20, 0x04, 0x00, 0x02, 0x00, 0x61, 0x00,
      0x08, 0xFF, 0x03, 0x2A, 0xDE, 0x00, 0x01, 0x0F,
      0x2A, 0xCE, 0x01, 0x1E, 0x01, 0x48, 0x00, 0x48,
      0x00, 0x01, 0x20, 0x10, 0x00, 0x20, 0x00, 0x00,
      0x00, 0x00, 0x48, 0x00, 0x48, 0x00, 0xFF, 0x00,
      0x10, 0xFF, 0x01, 0x81, 0xFF, 0x00, 0x10, 0xFF,
      0x01, 0x81, 0x2A, 0xD1, 0x02, 0x01, 0x07, 0x2A,
      0xDF, 0x02, 0x06, 0x04, 0x00, 0x00, 0x70, 0x81,
      0xAB, 0x2A, 0xCB, 0x03, 0x04, 0x98, 0x00, 0xC1,
      0xAC, 0x2A, 0xDF, 0x03, 0x37, 0x80, 0x04, 0x01,
      0x81, 0x7F, 0x01, 0x81, 0xFC, 0x00, 0x10, 0xFD,
      0x00, 0x10, 0x00, 0x07, 0x78, 0x01, 0x82, 0x07,
      0x00, 0x00, 0x04, 0x01, 0x82, 0xEE, 0x00, 0x20,
      0x00, 0xEE, 0x00, 0x20, 0x00, 0x48, 0x00, 0x05,
      0xAD, 0x81, 0xAC, 0x00, 0x00, 0x00, 0x00, 0x08,
      0x00, 0x98, 0x00, 0x00, 0x00, 0x98, 0x08, 0x00, 0x00,
      0x41, 0x9D, 0x81, 0xAC, 0x2A, 0xDB, 0x05, 0x05,
      0x90, 0x00, 0x90, 0x08, 0x80, 0x2A, 0xDF, 0x06,
      0x01, 0x04, 0x2A, 0x9F, 0x08, 0x01, 0x04, 0x2A,
      0x00
};
```

Note the #define at the top that defines the size of the array. This size is entered manually and is equal to the size given at the top of the text file (generated in 2.2 above) plus 6. This is because 6 dummy bytes have been added to the data, 5 at the start and 1 at the end.

Note also that “an_Byte” has been defined earlier in the software as an 8-bit integer using the “typedef” command (see code that accompanies this app note).

Now we have to write the code to send this data to the dpASP. The function below sends a single byte to the dpASP using a simple “bit-banging” technique. Note that the compiler used in this case was CCS. This compiler uses the key word “int8” to denote an 8-bit integer. Other compilers might use “char” or “unsigned char”.

```
void Data_Write(int8 data)
{
      int8 bit = 8;

      while (bit--)
      {
            if (data & 0x80)
                  DIN_High;
            DCLK_High;
            DCLK_Low;
            DIN_Low;
            data <<= 1;
      }
}
```

Some of the commands in this function have to be defined by the code writer e.g. DCLK_High. The code that is included with this app note was written for the AN221K04 development board. Refer to this code to see how DCLK_High, DIN_High etc were defined. These definitions are specific to the hardware so may need to be changed for a different application or board.

The code below does the actual primary configuration.

```
void Load_Primary_Circuit(void)
{
      int8 i;

      for (i = 0; i < Primary_Config_Size; i++)
            Data_Write(Primary_Data[i]);
}
```

3 Writing the Reconfiguration Code

3.1 Create the Reconfiguration Data

Go to the “Dynamic Config.” menu in AD2 and select “State-driven Method…”. Under the “States” tab select the primary state FPAA1 and click on “Remove” to delete it, then click on “Parameter Ranges”. Enter a name in the box at the top for the parameter that is to be varied (doesn’t matter what the name is but one must be entered.

Next enter 5 into the minimum oscillator frequency box and enter 25 into the maximum box (we will vary the oscillator frequency from 5kHz to 25kHz).

Next enter 4 into the number of steps (which means 5 states) and leave type as “Linear steps”.

Now press “OK”. Under the “Transitions” tab the various states will be seen.

Next select the “Generation” tab and click on “Generate” as before (same settings as before). Again a text file will be generated but this time it will contain 5 sets of reconfiguration data corresponding to the 5 oscillator frequencies 5, 10, 15, 20, 25kHz.

This data is shown below:

```
/*****************************************************************************\
*                                                                             *
*                          Transition Configurations                          *
*                                                                             *
\*****************************************************************************/

/*---------------------------------------------------------------------------*\
|                            osc: Fo[5] (34 bytes)                            |
\*---------------------------------------------------------------------------*/
0xD5, 0x01, 0x05, 0xC0, 0x04, 0x0A, 0x02, 0x01,
0x81, 0x40, 0x01, 0x81, 0xFE, 0x00, 0x10, 0xFE,
0x2A, 0xCE, 0x04, 0x01, 0x3C, 0x2A, 0x94, 0x04,
0x08, 0x02, 0x01, 0x82, 0xEE, 0x00, 0x20, 0x00,
0xEF, 0x2A, 0x00

/*---------------------------------------------------------------------------*\
|                            osc: Fo[10] (34 bytes)                           |
\*---------------------------------------------------------------------------*/
0xD5, 0x01, 0x05, 0xC0, 0x04, 0x0A, 0x04, 0x01,
0x81, 0x7F, 0x01, 0x81, 0xFC, 0x00, 0x10, 0xFD,
0x2A, 0xCE, 0x04, 0x01, 0x78, 0x2A, 0x94, 0x04,
0x08, 0x04, 0x01, 0x82, 0xEE, 0x00, 0x20, 0x00,
0xEE, 0x2A, 0x00

/*---------------------------------------------------------------------------*\
|                            osc: Fo[15] (34 bytes)                           |
\*---------------------------------------------------------------------------*/
0xD5, 0x01, 0x05, 0xC0, 0x04, 0x0A, 0x06, 0x01,
0x81, 0xBF, 0x01, 0x81, 0xFD, 0x00, 0x10, 0xFD,
0x2A, 0xCE, 0x04, 0x01, 0xB4, 0x2A, 0x94, 0x04,
0x08, 0x06, 0x01, 0x82, 0xEE, 0x00, 0x20, 0x00,
0xEF, 0x2A, 0x00

/*---------------------------------------------------------------------------*\
|                            osc: Fo[20] (34 bytes)                           |
\*---------------------------------------------------------------------------*/
0xD5, 0x01, 0x05, 0xC0, 0x04, 0x0A, 0x06, 0x01,
0x81, 0xBF, 0x01, 0x81, 0xBE, 0x00, 0x10, 0xBE,
0x2A, 0xCE, 0x04, 0x01, 0xD2, 0x2A, 0x94, 0x04,
0x08, 0x07, 0x01, 0x82, 0xD1, 0x00, 0x20, 0x00,
0xD1, 0x2A, 0x00

/*---------------------------------------------------------------------------*\
|                            osc: Fo[25] (34 bytes)                           |
\*---------------------------------------------------------------------------*/
0xD5, 0x01, 0x05, 0xC0, 0x04, 0x0A, 0x06, 0x01,
0x81, 0xBF, 0x01, 0x81, 0x98, 0x00, 0x10, 0x99,
0x2A, 0xCE, 0x04, 0x01, 0x96, 0x2A, 0x94, 0x04,
0x08, 0x05, 0x01, 0x82, 0x78, 0x00, 0x20, 0x00,
0x78, 0x2A, 0x00
```

Write the Reconfiguration Code

There is a lot of repetition in the data generated above, so rather than making 5 arrays of full reconfiguration data we can reduce it to the following functions:

```
#define UpdateGainSize                30
void Osc_Freq_Reconfig(int8 freq)
{
      an_Byte  Reconfig_Data[] = {0xC0, 0x04, 0x0A, 0x02, 0x01, 0x81, 0x40, 0x01, 0x81,
      0xFE, 0x00, 0x10, 0xFE, 0x2A, 0xCE, 0x04, 0x01, 0x3C, 0x2A, 0x94, 0x04, 0x08, 0x02,
      0x01, 0x82, 0xEE, 0x00, 0x20, 0x00, 0xEF};

      const an_Byte cap_C1[] = {0x02, 0x04, 0x06, 0x06, 0x06};
      const an_Byte cap_C2[] = {0x40, 0x7F, 0xBF, 0xBF, 0xBF};
      const an_Byte cap_C3[] = {0xFE, 0xFC, 0xFD, 0xBE, 0x98};
      const an_Byte cap_C4[] = {0xFE, 0xFD, 0xFD, 0xBE, 0x99};
      const an_Byte cap_C5[] = {0x3C, 0x78, 0xB4, 0xD2, 0x96};
      const an_Byte cap_C6[] = {0x02, 0x04, 0x06, 0x07, 0x05};
      const an_Byte cap_C7[] = {0xEE, 0xEE, 0xEE, 0xD1, 0x78};
      const an_Byte cap_C8[] = {0xEF, 0xEE, 0xEF, 0xD1, 0x78};

      Reconfig_Data[3] = cap_C1[freq];
      Reconfig_Data[6] = cap_C2[freq];
      Reconfig_Data[9] = cap_C3[freq];
      Reconfig_Data[12] = cap_C4[freq];
      Reconfig_Data[17] = cap_C5[freq];
      Reconfig_Data[22] = cap_C6[freq];
      Reconfig_Data[25] = cap_C7[freq];
      Reconfig_Data[29] = cap_C8[freq];

      Reconfig(Reconfig_Data, UpdateGainSize);
}

void Reconfig(int8 *array, int8 size)
{
      int8 i;

      Data_Write(0xD5);
      Data_Write(0x01);
      Data_Write(0x05);
      for (i = 0 ; i < size ; i++)
                        Data_Write(array[i]);
      Data_Write(0x2A);
      Data_Write(0x00);
}
```

Note that instead of writing 5 different arrays (containing a lot of the same bytes), the code above uses a single array and changes bytes within that array to make the different reconfigurations. This makes the code more efficient, especially when using large numbers of arrays.

The complete code is available with this document. If this code is compiled and loaded into an AN221K04 development board, it should load the primary circuit into the AN221E04 and then reconfigure the oscillator frequency at 1 second intervals.