THIS SPEC IS OBSOLETE

Spec No: 001-16833

Spec Title: SIGNAL MIXING WITH PSOC(R) SWITCHED CAPACITOR BLOCKS - AN16833

Sunset Owner: Kannan Sadasivam (QVS)

Replaced by: None

Signal Mixing with PSoC® Switched Capacitor Blocks

Author: Sampath Selvaraj
Associated Project: Yes
Associated Part Family: CY8C27443 (Project), CY8C27x43, CY8C29x66
Software Version: PSoC® Designer™ 4.4
Related Application Notes: AN2041, AN2223

If you have a question, or need help with this application note, contact the author at samp@cypress.com.

Routing analog signals inside a PSoC® device can be tricky because the desired connections are often not directly available. This application note describes some useful methods to allow configurations that seem impossible, such as the developed example of an eight channel (four channel stereo) audio mixer with adjustment potentiometers.

OBSOLETE

Contents

Introduction 1
All Blocks are not created equal 2
A Switched Capacitor PGA 2
Creating an Eight Channel Mixer 4
Reading Potentiometers without an ADC 4
Noise 5
Possibilities and Improvements 5
Summary 22
Worldwide Sales and Design Support 24

Introduction

A PSoC analog user module's input choices are determined by its location. Certain locations have access to Port 0 input multiplexers; others have direct access to the Port 2 inputs. Some only have access to certain adjacent blocks and their column's analog output bus.

Placement of an analog user module requires the correct type of target analog blocks: Continuous Time or Switched Capacitor. Therefore, not all analog user modules have access to a desired signal without another analog module to route it. In some cases – such as routing a Port 0 signal through a PGA to feed it to an ADC – this is a straightforward process.

However, after all four possible PGAs are used and additional simultaneous analog input signals are required, it is confusing to route these signals in, especially to where they are needed.

Some inputs that are typically read with an analog-to-digital converter, such as adjustment potentiometers, can be set up to be read by a timer user module. This frees up valuable analog resources. Some tricks are shown to convert a voltage into a charge time using a single digital pin for discharge and input (one pin per potentiometer). Here is a method to multiplex as many potentiometers as desired sequentially with a single-timer user module.

By configuring generic switched capacitor user modules to act as PGA buffers, an eight channel mixer with both stereo and mono outputs can be completely implemented within a single PSoC device with no external routing.

This mixer includes 15 adjustment potentiometers. There are potentiometers to adjust levels of each of the eight inputs and master adjustments for the left, right, and mono channels. Four more potentiometer inputs control the pre-amp levels of the PGAs in line with some inputs. This

allows the large range adjustment needed for microphones (or other small signal sources) on half of the inputs (specifically, inputs C and D on each side). The current design allows for a sixteenth adjustment potentiometer, but it does not yet have a use.

A serial TX user module is also included and is configured to output a formatted single-line output of all 15 gains. It outputs every couple sets of readings while adjustments are made, at a rate of 57,600 bps. This status display may be viewed with any standard serial terminal.

All Blocks are not created equal

Continuous Time (CT) analog blocks are easy to use: simple analog modules such as PGAs and Comparators go here. All four CT blocks have access to at least four Port 0 input pins (the two center blocks have extra multiplexers allowing connection to any of the eight Port 0 input pins), and the analog column clock has very little effect (if any) on operation.

All four CT blocks are the same and have similar designations; “ACBxy,” where x is the analog row (always 0) and y is the analog column number (0 to 3). These four blocks are along the top row of the analog section, shown in Figure 1.

Switched Capacitor (SC) analog blocks can give the PSoC many capabilities. These blocks can perform such a variety of real time functions that they were given a completely customizable user module, instead of separate modules for each function. This “Generic SCBlock” user module is found as the only entry under the “Generic” category.

SC blocks come in two varieties – Type C and Type D. Originally, in the CY8C25xxx/26xxx PSoC families, these were Type A and Type B. This is detailed in **AN2041**, “Understanding Analog Switched Capacitor Blocks.” Most, if not all of it, still applies to the newer CY8C27x43 and CY8C29x66 devices.

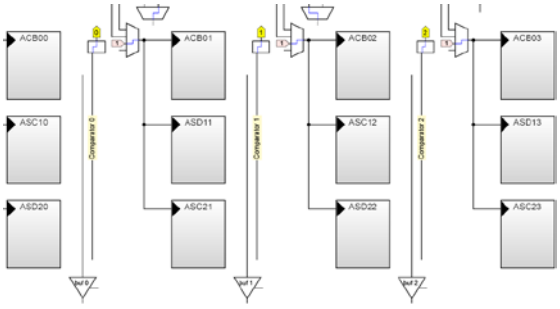
The old Type A blocks have evolved into the new Type C blocks and the old Type B blocks have evolved into the new type D blocks.

Type C SC blocks are designated “ASCxy,” where x is the analog row (1 or 2) and y is the analog column (0 to 3).

Type D SC blocks are designated “ASDxy,” where x is the analog row (1 or 2) and y is the analog column (0 to 3). Type D blocks include an additional option to allow BCap to function as a regular capacitor instead of a switched capacitor – set this BSW to “On” for normal operation.

Type C and D SC blocks are arranged in a checkerboard pattern, beginning with ASC10 as shown in Figure 1.

Figure 1. Analog Block Arrangement



A Switched Capacitor PGA

A Generic SC user module is configured to act as a PGA and amplify audio signals. The SC PGA does not have as large a range of gains as the standard CT PGA, but it is perfect for low and unity gain applications. The standard CT PGA user module provides gains from 0.062x to 48x, whereas the SC PGA created here has a range of 0.062x to only 1.938x (with an FCap value of 16; this range gets smaller and finer when FCap is set to 32). See Table 1 and Table 2 for details.

However, an SC PGA has the unique capability to mix two inputs and control their gains independently. An SC PGA can also invert its non-inverting input. The ratio between FCap and either ACap or BCap controls the gain applied to either the A or B inputs.

Empirically, an analog column clock of 4 MHz is optimum for an audio SC PGA. Faster clocks result in distortion, as the SC blocks do not perform quite as well near their upper limits.

The parameters necessary to create an SC PGA intended for use with AC signals (or audio) from a Generic SC user module are shown for the two types of SC blocks in Figure 2 and Figure 3.

Figure 2. SC Type C as a 2 Channel Audio PGA, 1.00x Gain

SC_LR

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	16
ACMux **(Input A)**	ASD20
BCap	16
AnalogBus	AnalogOutBus_1
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BMux **(Input B)**	ASD22
Power	High

OBSOLETE

Figure 3. SC Type D as a 2 Channel Audio PGA, 1.00x Gain

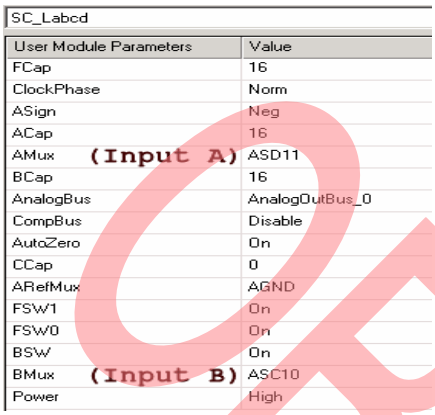


Table 1. SC PGA Gain Settings, FCap = 16

ACap or BCap	Gain
0	0
1	0.0625
2	0.125
3	0.1875
4	0.25
5	0.3125
6	0.375
7	0.4375
8	0.5
9	0.5625
10	0.625
11	0.6875
12	0.75
13	0.8125
14	0.875
15	0.9375
16	1
17	1.0625
18	1.125
19	1.1875
20	1.25
21	1.3125
22	1.375
23	1.4375
24	1.5
25	1.5625
26	1.625
27	1.6875
28	1.75
29	1.8125
30	1.875
31	1.9375

OBSOLETE

Table 2. SC PGA Gain Settings, FCap = 32

ACap or BCap	Gain
0	0
1	0.03125
2	0.06250
3	0.09375
4	0.12500
5	0.15625
6	0.18750
7	0.21875
8	0.25000
9	0.28125
10	0.31250
11	0.34375
12	0.37500
13	0.40625
14	0.43750
15	0.46875
16	0.50000
17	0.53125
18	0.56250
19	0.59375
20	0.62500
21	0.65625
22	0.68750
23	0.71875
24	0.75000
25	0.78125
26	0.81250
27	0.84375
28	0.87500
29	0.90625
30	0.93750
31	0.96875

Creating an Eight Channel Mixer

Using CT and SC PGA user modules, all eight simultaneously routable analog input signals (four CT inputs from Port 0 and four end-column SC inputs from Port 2) are mixed into three outputs — Left, Right, and Mono. The basic mixer schematic and internal routing is shown in Figure 5.

The Left output consists of a mix of input channels L_a, L_b, L_c, and L_d. The Right output consists of a mix of input channels R_a, R_b, R_c, and R_d. Mono output is a mix of all eight inputs.

Lc, Ld, Rc, and Rd are routed through CT PGAs before entering the mix and can have a pre-amp gain applied to them. This makes these channels ideal for use as microphone inputs.

The mixer portion of this project operates without logic. The only code required is to initialize the CT PGA modules; the SC PGA modules do not need start commands when their power level is set in the Device Editor to Low, Medium, or High.

If you can preset all your gain settings, then you are done. However, most applications require some real time control over these gains.

Reading Potentiometers without an ADC

Potentiometers are arranged into four banks of four inputs, requiring eight PSoC pins and one timer. Fifteen of the 16 possible potentiometer inputs from this configuration are used in the mixer design, as shown in Figure 6.

The four inputs to the PSoC are on P1[0], P1[1], P1[2], and P1[3]. The four bank control outputs are on P1[4], P1[5], P1[6], and P2[6].

The logic to read all 15 potentiometers sequentially is shown in the flowchart in Figure 7.

First, the potentiometer controlling the gain of L_a is serviced. P1[4] is raised to activate the first bank of inputs and the timer is set to capture the signal on P1[0]. Then, all four input pins are disconnected from their global input buses and changed to drive Open Drain Low outputs.

These outputs are driven low for a brief period to discharge the capacitors. They are then released, set back to High Z digital input drives, and reconnected to their global input busses. Then the timer is started.

A High Z digital input, whether connected to the global input bus or not, acts as an analog comparator referenced to ground with a threshold of approximately 0.500, or ½ of V_{CC}. This allows a timer to measure a potentiometer's position based on how long it takes its capacitor to charge up to this threshold level.

Now that the capacitor is discharged and the timer is started immediately afterwards (when the timer obtains a Capture event), its count is a direct ADC-style reading of the potentiometer. This is accessed after the interrupt through the timer's Compare register.

However, in some cases, the input threshold is reached before the command to start the timer is supplied. This is

OBSOLETE

not good, as timer user modules do not operate very well with a steady high capture signal. This situation incorrectly results in a minimum reading (zero). Special “Fast Charge” logic is implemented to catch this situation and handle it separately.

The timer is set up to produce a value from 0 to 127 for seven bits of resolution. These seven bits are shifted down to five bits, directly translating into ACap or BCap values for SC PGA gains.

In the case of the CT PGA gain adjustments, these five bits allow access to 32 of the available 33 gains. Because the top gain is so much more significant than the bottom gain, adding one to the CT PGA gain sets the top at 48x (and sets the bottom at 0.12x, excluding 0.06x).

This is done for simplicity and speed, as access to the bottom CT PGA gain is seldom required in this particular application. A separate algorithm (shifting the timer reading down to six bits to interpret the CT PGA gain) could be written if access to all 33 gains is required.

Figure 4. Example Serial Mixer Gain Display Output

```
 La | Lb || Lc | Ld | Ra | Rb | Rc | Rd | L  | R  | LR | MLc | MLd | MRc | MRd
1.13|0.00|0.00|0.75|1.13|0.00|0.00|0.75|1.13|0.00|0.00|01.33|48.00|00.12|00.87
```

Noise

The PSoC does not create isolated environments for all its modules and blocks. It also does not completely isolate the analog section from the digital section. As a result, many digital functions inject a small amount of audible noise into analog audio streams.

When the TX user module was initially set up to output on P2[7], starting the module (no transmissions) added audible noise into the otherwise quiet inputs on P2[0]–P2[3]. For this reason, the TX uses P1[7] as its output.

The scanning of the potentiometers also injects some unpleasant audible noise into the mix. This noise is much louder when any of the input signals are floating. If all eight inputs are connected, this noise is less noticeable.

The enable button or switch on P2[7] is implemented for this reason. The scanning of the potentiometers is only performed when P2[7] is high; it ceases when P2[7] is low. A button might be labeled “Hold to adjust or Push to update.” A switch might be labeled “Down to lock out changes and Up to adjust.”

There is another situation caused by leakage in this application. The bank select outputs were originally driven as Open Drain High, which should theoretically mix without diodes in the potentiometer scanning circuitry. The After the potentiometer for La is read (and the gain for La adjusted if there is a change), all bank select outputs are released. A short delay then creates a brief dead band.

This process is then repeated for the other seven input controls, three master controls, and four CT PGA pre-amp controls – each time activating the proper bank and reading the appropriate input to access the desired potentiometer.

Every three sets of readings, the applied gains of each of the 15 adjustments are sent out the serial TX port for display on a standard terminal. This allows the user to see exactly what gains are applied, and where. Example output is shown in Figure 4.

Configuration of the PSoC eight channel mixer is shown in Figure 8, Figure 9, Figure 10, and Figure 11 on page 14, 15, 16, and 19 respectively.

bank selects were driven high to select a particular bank, and released to float otherwise. Unfortunately, when floating, the bank outputs sharing Port 1 with the inputs were affected by the various charging capacitors. This resulted in multiple gains being adjusted unintentionally.

For this reason, the bank selects are now strong outputs. They are driven high to select a bank, and low otherwise. To ensure that low drives do not affect the potentiometer readings, blocking diodes (D_1, D_2, D_3, and D_4) are now required on these lines.

Notes

- A simple resistor divider provides the $V_{CC}/2$ AGND, used to bias the AC-coupled inputs.
- The two 100 µF bypass caps (C_{14}, C_{15}) are arranged in series with the supply to facilitate a quick initial charge-up.
- The ACap and BCap settings in the Device Editor have no effect because they are set in the software.

Possibilities and Improvements

The current design requires two programming jumpers to be removed to burn the PSoC in-circuit. This is because P1[0] and P1[1], which are the pins used for programming, have capacitance attached to them. If these capacitors are not removed when attempting in-circuit programming, it

OBSOLETE

fails. By exchanging these inputs with bank control signals and making the necessary changes to the software, in-circuit programming no longer requires jumpers.

A serial RX user module could be added to allow remote control of the gains. This may be useful in studio situations where multiple users in multiple locations might want to adjust the gains of a single mixer.

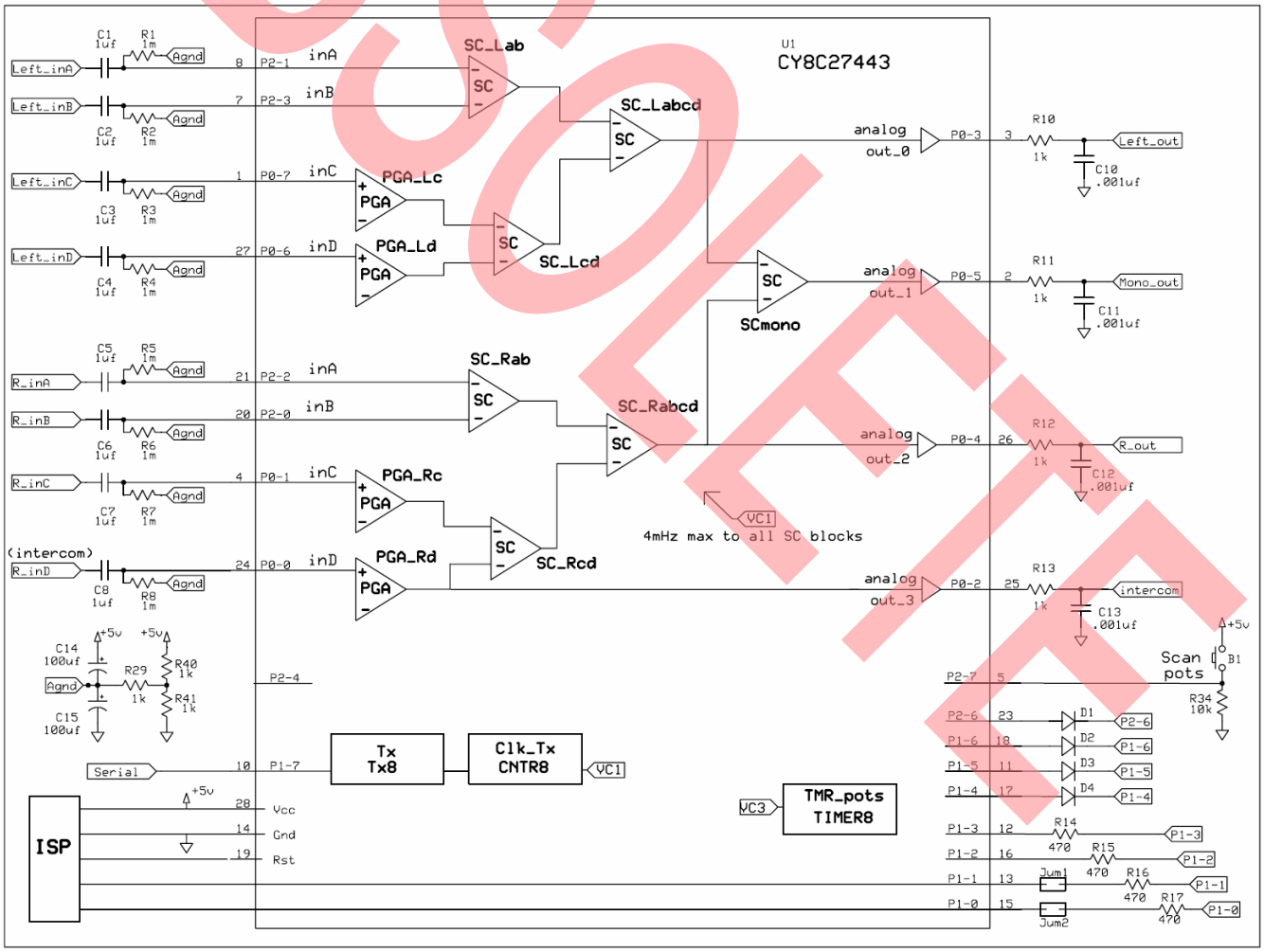
A slightly altered version of the eight channel mixer can act as a studio headphone amp, where the artist can select his or her own personal mix of seven line feeds along with an intercom channel back to the control room. PGA_Rd is configured as an intercom with its own output (P0[2]) and can easily be disconnected from SC_Rcd, if necessary.

Even with all the analog functionality present in this eight channel mixer, there is still a single SC block left open. Another analog feature can also be added to this project.

As shown in **AN2412**, “Reverb and Echo with a PSoC-Style FIFO,” a DelSig ADC user module may be used to sample a stream of audio. The single-order DelSig only requires a single SC block (and a digital block), and has its choice of input signals.

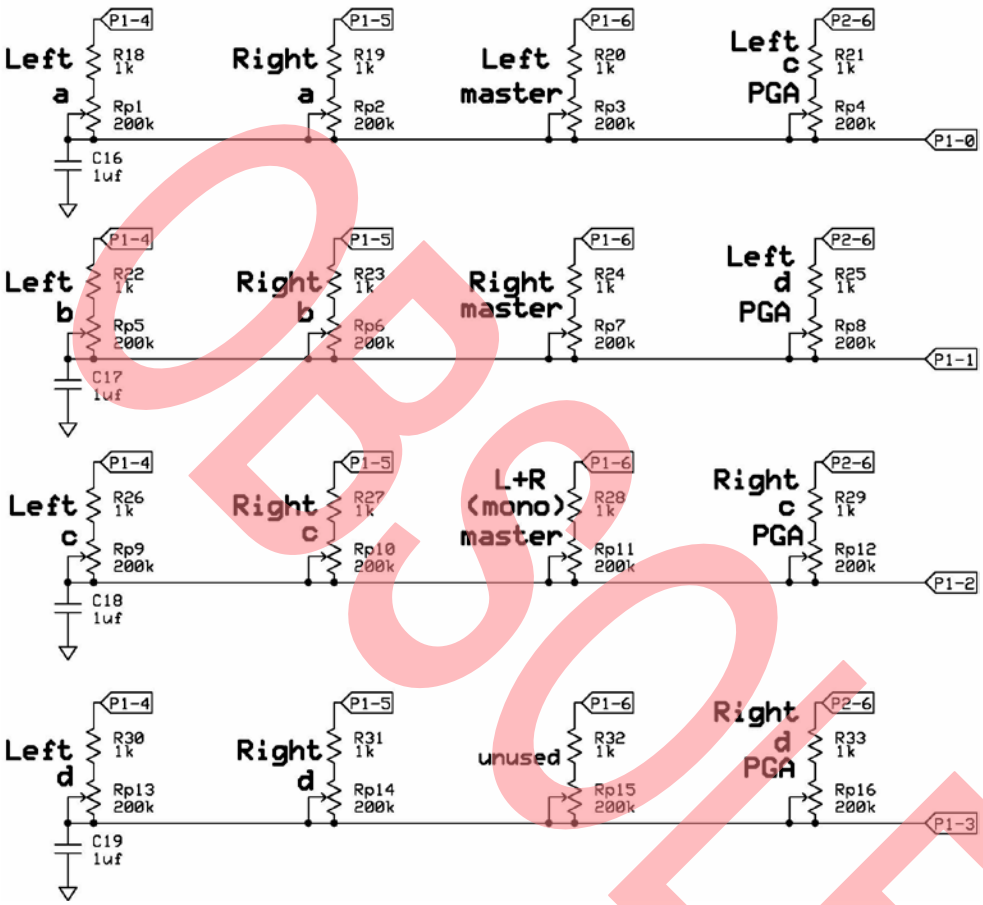
With the PSoC adding a bit of reverb and a PRS8 user module representing this altered stream, there may be use for the 16th potentiometer input – adjusting the amount of reverb or echo. This creates another studio application. A singer can control his or her own “scratch mix wetness,” as personal unrecorded reverb adjustments are referred to in the musicians’ world.

Figure 5. Audio Mixer Schematic



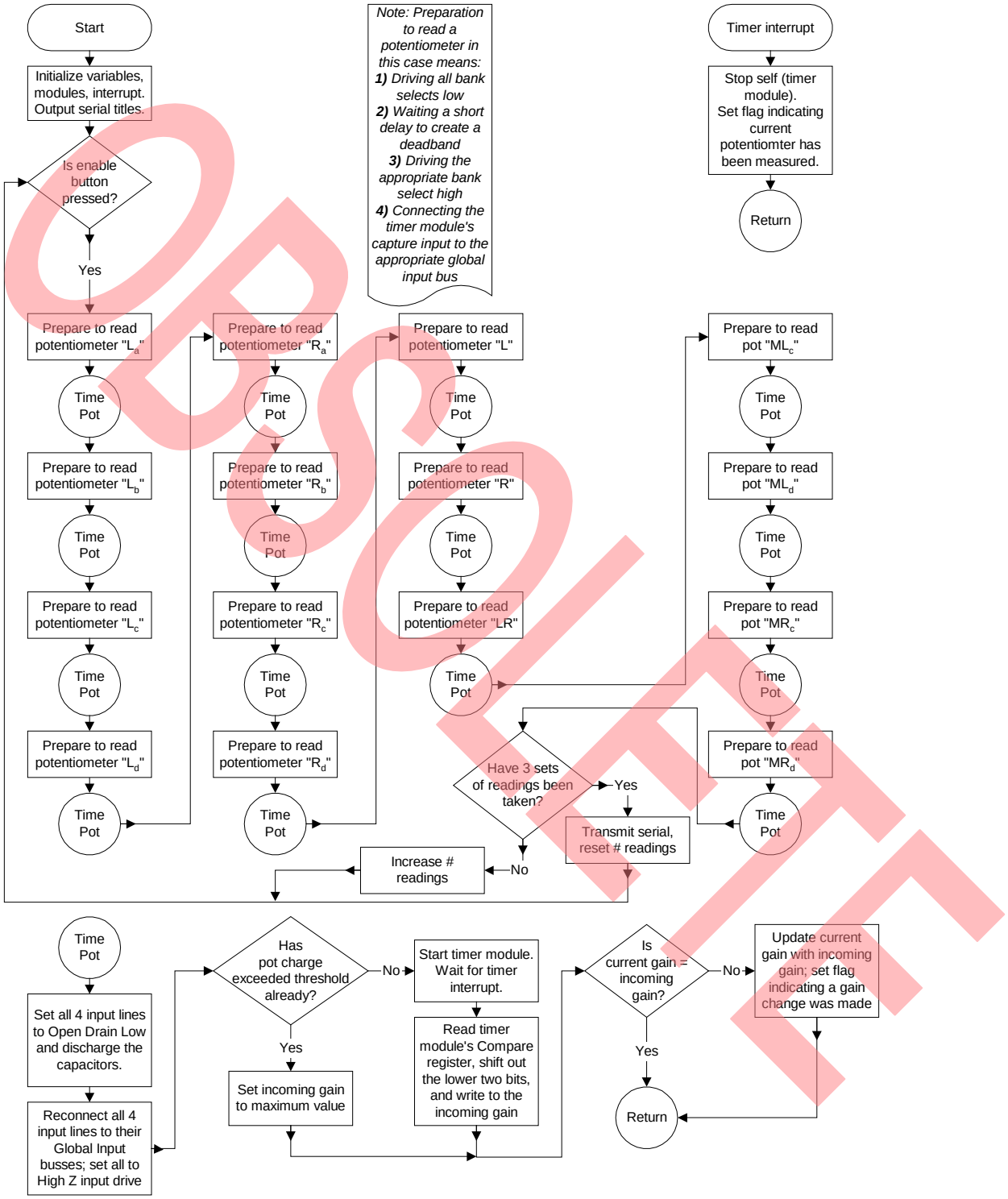
OBSOLETE

Figure 6. Potentiometer/Knob Interface Schematic



OBSOLETE

Figure 7. Eight Channel Mixer Flowchart



OBSOLETE

Code 1. Eight Channel Mixer Source Code

```
//----------main.c:
//-------------------------------------------------------------------------------
#define CyclesToDischargeKnob 50        //0-4million:  number  of  cycles  to  wait  whilst
discharging each knob
#define CyclesOfDeadbandBetweenReadings 100    //0-4million:  number  of  cycles  to  wait  in
between readings to enable settling

#define SerialOutputEnabled 1           //Comment out this line to disable the serial output
function
#define ReadingSetsBetweenSerialBursts 3       //0-255

//-------------------------------------------------------------------------------
#include <m8c.h>        // part specific constants and macros
#include "PSoCAPI.h"    // PSoC API definitions for all User Modules
#include <stdlib.h>
//-------------------------------------------------------------------
#define Bit(bitNumber) ( 1 << (bitNumber) )
unsigned long int Counter = 0;
void Wait(unsigned long int);
void Wait(unsigned long int ToWait){ for (Counter = 0; Counter < ToWait; Counter++);   }
//-------------------------------------------------------------
#define OutputSerialTitles TX_CPutString(     " La | Lb | Lc | Ld | Ra | Rb | Rc | Rd | L  |
R  | LR | MLc | MLd | MRc | MRd ")
//
   "1.00|1.00|1.00|1.00|1.00|1.00|1.00|1.00|1.00|1.00|1.00|48.00|48.00|48.00|48.00"
#define OutputSerialSeparator TX_PutChar('|')
//-------------------------------------------------------------------------------------------
char ZTS[9];
//void CharToZTS(unsigned char);
//void CharToZTS(unsigned char ToConvert)
//{
// if (ToConvert >= 100) itoa(&ZTS[0], ToConvert, 10);
// else if (ToConvert >= 10){ZTS[0]='0'; itoa(&ZTS[1], ToConvert, 10);}
// else{ZTS[0]='0'; ZTS[1]='0'; itoa(&ZTS[2], ToConvert, 10);}
// ZTS[3] = 0;      //for ZTS
//}

unsigned char Index=0;
unsigned char Target=0;
void IntToFloatZTS(unsigned int, unsigned char);
void IntToFloatZTS(unsigned int ToConvert, unsigned char NumDecimalPlaces)
{
   if (ToConvert >= 10000) itoa(&ZTS[0], ToConvert, 10);
   else if (ToConvert >= 1000){ZTS[0]='0';    itoa(&ZTS[1], ToConvert, 10);}
   else if (ToConvert >= 100){ZTS[0]='0'; ZTS[1]='0'; itoa(&ZTS[2], ToConvert, 10);}
   else  if  (ToConvert  >=  10){ZTS[0]='0';  ZTS[1]='0';  ZTS[2]='0';  itoa(&ZTS[3],  ToConvert,
10);}
   else{ZTS[0]='0'; ZTS[1]='0'; ZTS[2]='0'; ZTS[3]='0'; itoa(&ZTS[4], ToConvert, 10);}
   ZTS[5] = 0;      //for ZTS
   if (NumDecimalPlaces)
   {
      Target=5-NumDecimalPlaces;       //init target to where decimal point should be
      Index=6;
      while (Index>Target)
      {
         ZTS[Index]=ZTS[Index-1];            //shift char right
         Index--;
      }
```

OBSOLETE

```
        ZTS[Index]='.';                 //insert decimal point
    }
}

void TransmitSerial(void);
//-------------------------------------------------------------------------------------
//-----CT PGA
#define NumGains 33
const BYTE CtPgaGains[NumGains]={PGA_Lc_G0_06,PGA_Lc_G0_12,PGA_Lc_G0_18,PGA_Lc_G0_25,  //0-3
    PGA_Lc_G0_31,PGA_Lc_G0_37,PGA_Lc_G0_43,PGA_Lc_G0_50,PGA_Lc_G0_56,PGA_Lc_G0_62,     //4-9
    PGA_Lc_G0_68,PGA_Lc_G0_75,PGA_Lc_G0_81,PGA_Lc_G0_87,PGA_Lc_G0_93,PGA_Lc_G1_00,    //10-15
    PGA_Lc_G1_06,PGA_Lc_G1_14,PGA_Lc_G1_23,PGA_Lc_G1_33,PGA_Lc_G1_46,PGA_Lc_G1_60,    //16-21
    PGA_Lc_G1_78,PGA_Lc_G2_00,PGA_Lc_G2_27,PGA_Lc_G2_67,PGA_Lc_G3_20,PGA_Lc_G4_00,    //22-27
PGA_Lc_G5_33,PGA_Lc_G8_00,PGA_Lc_G16_0,PGA_Lc_G24_0,PGA_Lc_G48_0};                    //28-32
const                                   unsigned                                    int
CtPgaGainValues[NumGains]={6,12,18,25,31,37,43,50,56,62,68,75,81,87,93,100,106,114,123,     \
133,146,160,178,200,227,267,320,400,533,800,1600,2400,4800};
//index 15 = 1.00x
#define CtPgaGainOffset 1  //shift incoming gain value this far up (add this) to ensure
access          //to 24x and 48x gains (skips 0.06x and 0.12x though)
#define DefaultCtPgaGain 15
//-----SC PGA
#define NumScPgaGains 32

//The below constant contains the SC PGA gain values in fixed-point decimal (to avoid float
values) - //move decimal point two digits left for real value and display
//Below constant only needed for displaying gain values on an LCD (for example)
//For reference, the below values represent gains 0.00x - 1.94x, varying ACap and BCap
equally while //holding FCap constant at 16
const                                   unsigned                                    int
ScPgaGainValuesFCap16[NumScPgaGains]={0,6,13,19,25,31,38,44,50,56,62,69,75,81,88,94,100,106,1
13,119,125,131,138,144,150,156,162,169,175,181,188,194};

#define DefaultScPgaGain 0
#define MaxScPgaGain NumScPgaGains-1
#define MinScPgaGain 0
/*
CR0 [43210] = ACap
 i.e. SC_Lab_cr0

CR1 [43210] = BCap
 i.e. SC_Lab_cr1
*/

unsigned char TempCr;
unsigned char TempCr1;
#define ResetAorBCap 0b11100000
unsigned char IncomingGain;
//Bank0
unsigned char GainLab_A=DefaultScPgaGain;
unsigned char GainLab_B=DefaultScPgaGain;
unsigned char GainLcd_C=DefaultScPgaGain;
unsigned char GainLcd_D=DefaultScPgaGain;
#define UpdateLab_AGain TempCr=(SC_Lab_cr0 & ResetAorBCap) | GainLab_A; SC_Lab_cr0=TempCr
#define UpdateLab_BGain TempCr=(SC_Lab_cr1 & ResetAorBCap) | GainLab_B; SC_Lab_cr1=TempCr
#define UpdateLcd_CGain TempCr=(SC_Lcd_cr0 & ResetAorBCap) | GainLcd_C; SC_Lcd_cr0=TempCr
#define UpdateLcd_DGain TempCr=(SC_Lcd_cr1 & ResetAorBCap) | GainLcd_D; SC_Lcd_cr1=TempCr
//Bank1
unsigned char GainRab_A=DefaultScPgaGain;
unsigned char GainRab_B=DefaultScPgaGain;
```

OBSOLETE

```
unsigned char GainRcd_C=DefaultScPgaGain;
unsigned char GainRcd_D=DefaultScPgaGain;
#define UpdateRab_AGain TempCr=(SC_Rab_cr0 & ResetAorBCap) | GainRab_A; SC_Rab_cr0=TempCr
#define UpdateRab_BGain TempCr=(SC_Rab_cr1 & ResetAorBCap) | GainRab_B; SC_Rab_cr1=TempCr
#define UpdateRcd_CGain TempCr=(SC_Rcd_cr0 & ResetAorBCap) | GainRcd_C; SC_Rcd_cr0=TempCr
#define UpdateRcd_DGain TempCr=(SC_Rcd_cr1 & ResetAorBCap) | GainRcd_D; SC_Rcd_cr1=TempCr
//Bank2
unsigned char GainL=DefaultScPgaGain;
unsigned char GainR=DefaultScPgaGain;
unsigned char GainLR=DefaultScPgaGain;
#define UpdateLGain TempCr=(SC_Labcd_cr0 & ResetAorBCap) | GainL; TempCr1=(SC_Labcd_cr1 &
ResetAorBCap) | GainL; SC_Labcd_cr0=TempCr; SC_Labcd_cr1=TempCr1
#define UpdateRGain TempCr=(SC_Rabcd_cr0 & ResetAorBCap) | GainR; TempCr1=(SC_Rabcd_cr1 &
ResetAorBCap) | GainR; SC_Rabcd_cr0=TempCr; SC_Rabcd_cr1=TempCr1
#define UpdateLRGain TempCr=(SC_LR_cr0 & ResetAorBCap) | GainLR; TempCr1=(SC_LR_cr1 &
ResetAorBCap) | GainLR; SC_LR_cr0=TempCr; SC_LR_cr1=TempCr1
//Mic gains are CT PGA gains, Bank3
unsigned char GainMicLc=DefaultCtPgaGain;
unsigned char GainMicLd=DefaultCtPgaGain;
unsigned char GainMicRc=DefaultCtPgaGain;
unsigned char GainMicRd=DefaultCtPgaGain;
#define UpdateMicLcGain PGA_Lc_SetGain(CtPgaGains[GainMicLc+CtPgaGainOffset])
#define UpdateMicLdGain PGA_Ld_SetGain(CtPgaGains[GainMicLd+CtPgaGainOffset])
#define UpdateMicRcGain PGA_Rc_SetGain(CtPgaGains[GainMicRc+CtPgaGainOffset])
#define UpdateMicRdGain PGA_Rd_SetGain(CtPgaGains[GainMicRd+CtPgaGainOffset])

unsigned char KnobReading=0;
//-------------------------------------------------------------------------------------
#define       ScaleReadingOfKnob       KnobReading>>=2;       if(KnobReading>MaxScPgaGain){
KnobReading=MaxScPgaGain; }
#define KNOB_FASTCHARGE 3
#define KNOB_SERVICED 2
#define KNOB_TIMED 1
#define KNOB_TIMING 0
unsigned char KnobStatus=0;
//Target charge rate of ~188Hz -- ok, adjusted to 178Hz to match 7 bits of resolution to
shift directly down to 5 for ACap/BCap control
#define IsKnobTimed (KnobStatus==KNOB_TIMED)
#define IsKnobServiced (KnobStatus==KNOB_SERVICED)
#define IsKnobFastCharge (KnobStatus==KNOB_FASTCHARGE)
#define SetKnobAsServiced KnobStatus=KNOB_SERVICED
#define SetKnobAsTimed KnobStatus=KNOB_TIMED
#define SetKnobAsFastCharge KnobStatus=KNOB_FASTCHARGE
#define ResetKnobStatus KnobStatus=KNOB_TIMING
#define ResetKnobTmr TMR_Knob_WritePeriod(TMR_Knob_PERIOD); TMR_Knob_WriteCompareValue(0)
#define ReadKnobTmr KnobReading=TMR_Knob_COMPARE_REG; \
   ScaleReadingOfKnob; IncomingGain=KnobReading
#define ServiceKnob if(IsKnobTimed) \
   { \
      ReadKnobTmr; \
      ResetKnobTmr; \
      SetKnobAsServiced; \
   } \
   else if(IsKnobFastCharge) \
   { \
      IncomingGain=MaxScPgaGain; \
      ResetKnobTmr; \
      SetKnobAsServiced; \
   }
#define CheckChargeStatusRowIn0 (Bit(0))
```

OBSOLETE

```
#define CheckChargeStatusRowIn1 (Bit(1))
#define CheckChargeStatusRowIn2 (Bit(2))
#define CheckChargeStatusRowIn3 (Bit(3))
unsigned char CurrentInputMask=CheckChargeStatusRowIn1;
#define CheckCurrentChargeStatus (PRT1DR&CurrentInputMask)
#define     AllInputsMask     (CheckChargeStatusRowIn0     |     CheckChargeStatusRowIn1     |
CheckChargeStatusRowIn2 | CheckChargeStatusRowIn3)
#define      AllInputsToOpenDrainLow      PRT1DM2|=AllInputsMask;      PRT1DM1|=AllInputsMask;
PRT1DM0|=AllInputsMask
#define         AllInputsToHiZ        PRT1DM2&=~AllInputsMask;        PRT1DM1|=AllInputsMask;
PRT1DM0&=~AllInputsMask
#define AllInputsToStdCpu PRT1GS&=~AllInputsMask
#define AllInputsToGlobalIO PRT1GS|=AllInputsMask
#define DischargeKnobInput PRT1DR&=~AllInputsMask
#define ReleaseKnobInput PRT1DR|=AllInputsMask
#define  DrainAndStartKnob  AllInputsToStdCpu;  AllInputsToOpenDrainLow;  DischargeKnobInput;
Wait(CyclesToDischargeKnob); \
    ResetKnobStatus; ReleaseKnobInput; AllInputsToHiZ; \
    if(CheckCurrentChargeStatus) { SetKnobAsFastCharge; } \
    else { TMR_Knob_Start(); ResetKnobStatus; } \
    AllInputsToGlobalIO
#define CheckKnobEnableButton (PRT2DR&Bit(7))
//------------------------------------------------------------------------
//P1_4, P1_5, P1_6 = KnobBank0, 1, 2
//P2_6=KnobBank3
#define    AllBanksOff    PRT1DR&=~(Bit(4)    |    Bit(5)    |    Bit(6));    PRT2DR&=~(Bit(6));
Wait(CyclesOfDeadbandBetweenReadings)
#define KnobToBank0 AllBanksOff; PRT1DR|=Bit(4)
#define KnobToBank1 AllBanksOff; PRT1DR|=Bit(5)
#define KnobToBank2 AllBanksOff; PRT1DR|=Bit(6)
#define KnobToBank3 AllBanksOff; PRT2DR|=Bit(6)

#define KnobInputToRow0In TMR_Knob_INPUT_REG=0xc1; CurrentInputMask=CheckChargeStatusRowIn0
#define KnobInputToRow1In TMR_Knob_INPUT_REG=0xd1; CurrentInputMask=CheckChargeStatusRowIn1
#define KnobInputToRow2In TMR_Knob_INPUT_REG=0xe1; CurrentInputMask=CheckChargeStatusRowIn2
#define KnobInputToRow3In TMR_Knob_INPUT_REG=0xf1; CurrentInputMask=CheckChargeStatusRowIn3


//Left A,B,C,D input adjustments on Bank0
#define PrepareToReadKnobLab_A KnobToBank0; KnobInputToRow0In
#define PrepareToReadKnobLab_B KnobToBank0; KnobInputToRow1In
#define PrepareToReadKnobLcd_C KnobToBank0; KnobInputToRow2In
#define PrepareToReadKnobLcd_D KnobToBank0; KnobInputToRow3In

//Right A,B,C,D input adjustments on Bank1
#define PrepareToReadKnobRab_A KnobToBank1; KnobInputToRow0In
#define PrepareToReadKnobRab_B KnobToBank1; KnobInputToRow1In
#define PrepareToReadKnobRcd_C KnobToBank1; KnobInputToRow2In
#define PrepareToReadKnobRcd_D KnobToBank1; KnobInputToRow3In

//Master L, R, LR input adjustments on Bank2
#define PrepareToReadKnobL KnobToBank2; KnobInputToRow0In
#define PrepareToReadKnobR KnobToBank2; KnobInputToRow1In
#define PrepareToReadKnobLR KnobToBank2; KnobInputToRow2In

//Mic Preamp input adjustments on Bank3
#define PrepareToReadKnobMicLc KnobToBank3; KnobInputToRow0In
#define PrepareToReadKnobMicLd KnobToBank3; KnobInputToRow1In
#define PrepareToReadKnobMicRc KnobToBank3; KnobInputToRow2In
#define PrepareToReadKnobMicRd KnobToBank3; KnobInputToRow3In
```

OBSOLETE

```
BOOL GainChangeThisCycle=0;
#define     CycleThroughAndReadTheInputs     PrepareToReadKnobLab_A;     DrainAndStartKnob;
while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainLab_A){           GainLab_A=IncomingGain;           UpdateLab_AGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobLab_B; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainLab_B){           GainLab_B=IncomingGain;           UpdateLab_BGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobLcd_C; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainLcd_C){           GainLcd_C=IncomingGain;           UpdateLcd_CGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobLcd_D; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainLcd_D){           GainLcd_D=IncomingGain;           UpdateLcd_DGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobRab_A; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainRab_A){           GainRab_A=IncomingGain;           UpdateRab_AGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobRab_B; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainRab_B){           GainRab_B=IncomingGain;           UpdateRab_BGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobRcd_C; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainRcd_C){           GainRcd_C=IncomingGain;           UpdateRcd_CGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobRcd_D; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainRcd_D){           GainRcd_D=IncomingGain;           UpdateRcd_DGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobL; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainL){ GainL=IncomingGain; UpdateLGain; GainChangeThisCycle=1; } \
   PrepareToReadKnobR; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainR){ GainR=IncomingGain; UpdateRGain; GainChangeThisCycle=1; } \
   PrepareToReadKnobLR; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainLR){ GainLR=IncomingGain; UpdateLRGain; GainChangeThisCycle=1; } \
   PrepareToReadKnobMicLc; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainMicLc){           GainMicLc=IncomingGain;           UpdateMicLcGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobMicLd; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainMicLd){           GainMicLd=IncomingGain;           UpdateMicLdGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobMicRc; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainMicRc){           GainMicRc=IncomingGain;           UpdateMicRcGain;
GainChangeThisCycle=1; } \
   PrepareToReadKnobMicRd; DrainAndStartKnob; while(!IsKnobServiced){ ServiceKnob; } \
   if(IncomingGain!=GainMicRd){           GainMicRd=IncomingGain;           UpdateMicRdGain;
GainChangeThisCycle=1; } \
   AllBanksOff

//--
unsigned char SerialOutputCounter=0;
//------------------------------------------------------------------------------------

void main()
{
   UpdateLab_AGain;
   UpdateLab_BGain;
   UpdateLcd_CGain;
   UpdateLcd_DGain;
   UpdateRab_AGain;
   UpdateRab_BGain;
   UpdateRcd_CGain;
```

PROOF IT

```
    UpdateRcd_DGain;


UpdateLGain;
    UpdateRGain;
    UpdateLRGain;
    UpdateMicLcGain;
    UpdateMicLdGain;
    UpdateMicRcGain;
    UpdateMicRdGain;
     PGA_Lc_Start(PGA_Lc_HIGHPOWER);
     PGA_Ld_Start(PGA_Ld_HIGHPOWER);
     PGA_Rc_Start(PGA_Rc_HIGHPOWER);
     PGA_Rd_Start(PGA_Rd_HIGHPOWER);
    Clk_Tx_Start();
    TX_Start(TX_PARITY_NONE);
    TMR_Knob_EnableInt();
     M8C_EnableGInt;
     TX_PutCRLF();
     TX_PutCRLF();
     OutputSerialTitles;
     TX_PutCRLF();

     while(1)
     {
        if(CheckKnobEnableButton)
        {
            CycleThroughAndReadTheInputs;
            if(++SerialOutputCounter>ReadingSetsBetweenSerialBursts)
            {
                if(GainChangeThisCycle){ TransmitSerial(); GainChangeThisCycle=0; }
                SerialOutputCounter=0;
            }
        }
     }
}

void TransmitSerial(void)
{
#ifdef SerialOutputEnabled
    IntToFloatZTS(ScPgaGainValuesFCap16[GainLab_A],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainLab_B],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainLcd_C],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainLcd_D],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainRab_A],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainRab_B],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainRcd_C],2);
    TX_PutString(&ZTS[2]);
```

OBSOLETE

```
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainRcd_D],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainL],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainR],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(ScPgaGainValuesFCap16[GainLR],2);
    TX_PutString(&ZTS[2]);
    OutputSerialSeparator;
    IntToFloatZTS(CtPgaGainValues[GainMicLc+CtPgaGainOffset],2);

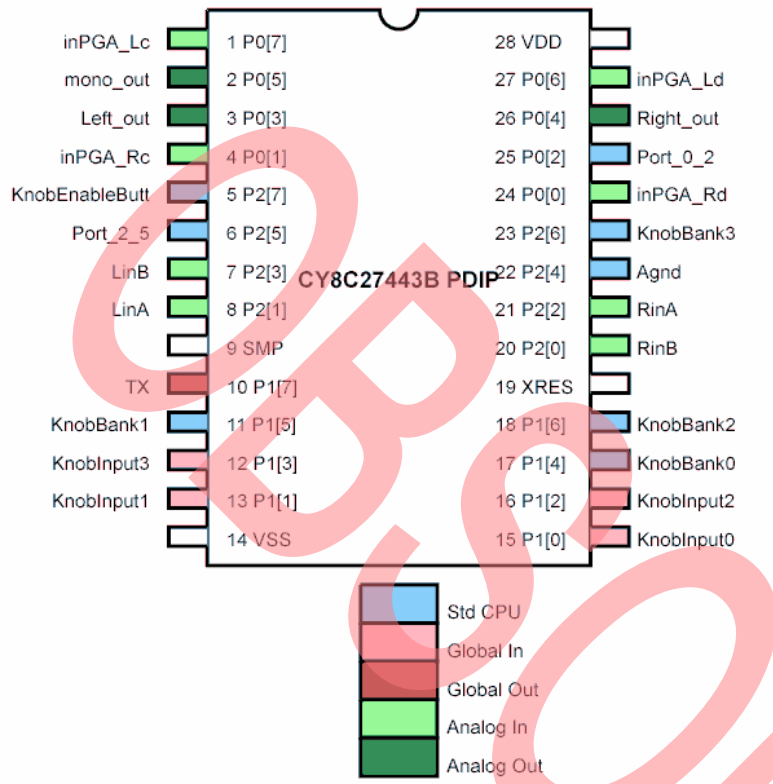

TX_PutString(&ZTS[1]);
    OutputSerialSeparator;
    IntToFloatZTS(CtPgaGainValues[GainMicLd+CtPgaGainOffset],2);
    TX_PutString(&ZTS[1]);
    OutputSerialSeparator;
    IntToFloatZTS(CtPgaGainValues[GainMicRc+CtPgaGainOffset],2);
    TX_PutString(&ZTS[1]);
    OutputSerialSeparator;
    IntToFloatZTS(CtPgaGainValues[GainMicRd+CtPgaGainOffset],2);
    TX_PutString(&ZTS[1]);
    TX_PutChar(13);
#endif
}


//----------TMR_KnobINT.asm:
_TMR_Knob_ISR:

    ;@PSoC_UserCode_BODY@ (Do not change this line.)
    ;---------------------------------------------------
    ; Insert your custom code below this banner
    ;---------------------------------------------------
    ;   NOTE: interrupt service routines must preserve
    ;   the values of the A and X CPU registers.
push A
    TMR_Knob_Stop_M
    mov [_KnobStatus], 1
pop A
```

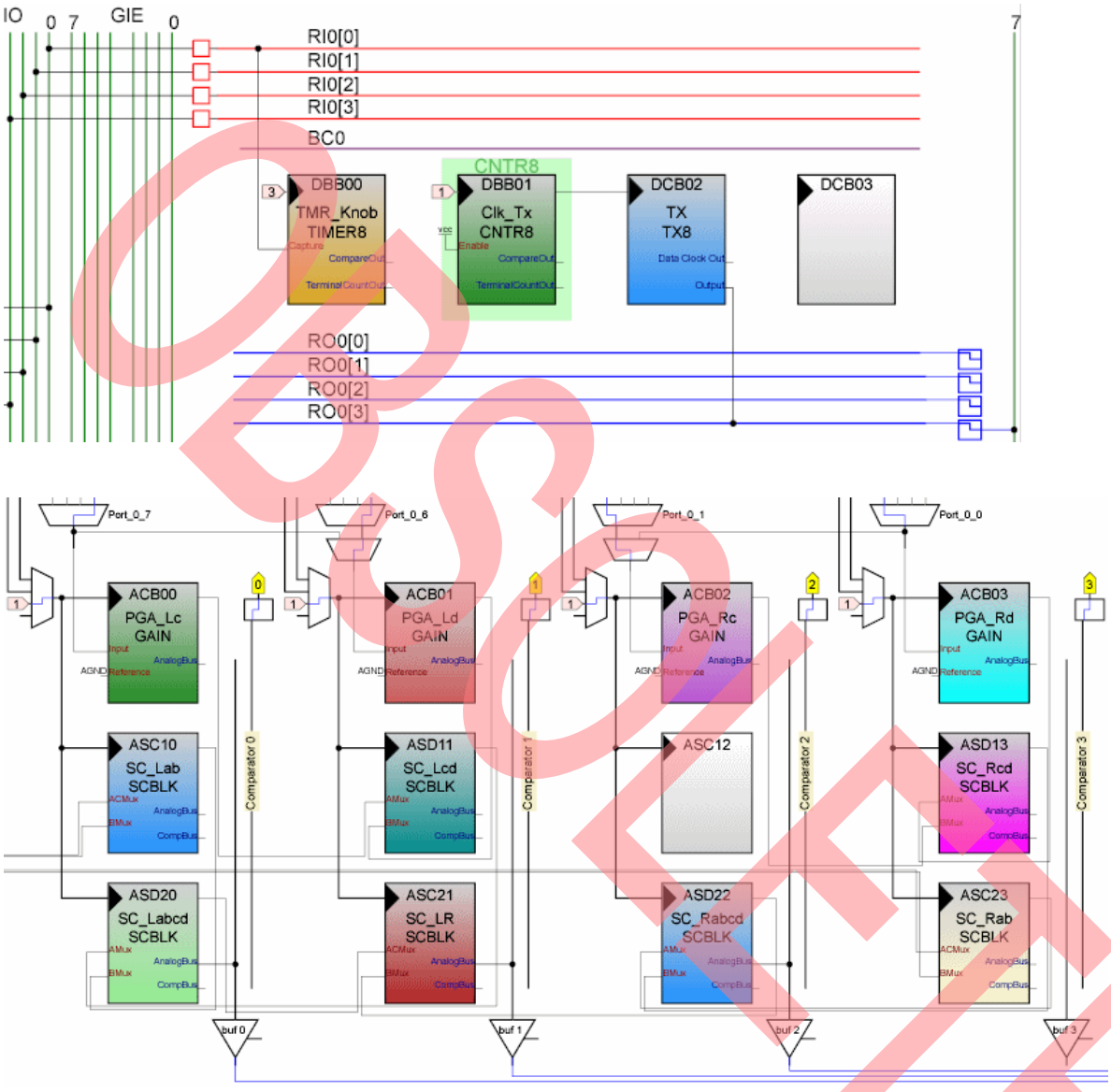
OBSOLETE

Figure 8. Eight Channel Mixer PSoC Pinout Configuration



OBSOLETE

Figure 9. Eight Channel Mixer PSoC User Module Placement GUI



OBSOLETE

Figure 10. Eight Channel Mixer User Module Parameter Configuration

Clk_Tx

User Module Parameters	Value
Clock	VC1
ClockSync	Use SysClk Direct
Enable	High
CompareOut	None
TerminalCountOut	None
Period	51
CompareValue	1
CompareType	Less Than Or Equal
InterruptType	Terminal Count
InvertEnable	Normal

PGA_Ld

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputSelect_1
Reference	AGND
AnalogBus	Disable

PGA_Lc

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputMUX_0
Reference	AGND
AnalogBus	Disable

PGA_Rc

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputSelect_2
Reference	AGND
AnalogBus	Disable

PGA_Rd

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputMUX_3
Reference	AGND
AnalogBus	Disable

SC_Lab

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	0
ACMux	Port_2_1
BCap	0
AnalogBus	Disable
CompBus	Disable
AutoZero	On
CCap	16
ARefMux	AGND
FSW1	On
FSW0	On
BMux	Port_2_3
Power	High

SC_Labcd

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	16
AMux	ASD11
BCap	16
AnalogBus	AnalogOutBus_0
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BSW	On
BMux	ASC10
Power	High

OBSOLETE

SC_Lcd

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	0
AMux	ACB00
BCap	0
AnalogBus	Disable
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BSW	On
BMux	ACB01
Power	High

SC_LR

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	16
ACMux	ASD20
BCap	16
AnalogBus	AnalogOutBus_1
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BMux	ASD22
Power	High

SC_Rab

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	0
ACMux	Port_2_2
BCap	0
AnalogBus	Disable
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BMux	Port_2_0
Power	High

SC_Rabcd

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	16
AMux	ASC23
BCap	16
AnalogBus	AnalogOutBus_2
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BSW	On
BMux	ASD13
Power	High

OBSOLETE

Clk_Tx

User Module Parameters	Value
Clock	VC1
ClockSync	Use SysClk Direct
Enable	High
CompareOut	None
TerminalCountOut	None
Period	51
CompareValue	1
CompareType	Less Than Or Equal
InterruptType	Terminal Count
InvertEnable	Normal

PGA_Rc

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputSelect_2
Reference	AGND
AnalogBus	Disable

SC_Lab

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	0
ACMux	Port_2_1
BCap	0
AnalogBus	Disable
CompBus	Disable
AutoZero	On
CCap	16
ARefMux	AGND
FSW1	On
FSW0	On
BMux	Port_2_3
Power	High

PGA_Ld

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputSelect_1
Reference	AGND
AnalogBus	Disable

PGA_Lc

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputMUX_0
Reference	AGND
AnalogBus	Disable

PGA_Rd

User Module Parameters	Value
Gain	1.000
Input	AnalogColumn_InputMUX_3
Reference	AGND
AnalogBus	Disable

SC_Labcd

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	16
AMux	ASD11
BCap	16
AnalogBus	AnalogOutBus_0
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BSW	On
BMux	ASC10
Power	High

OBSOLETE

SC_Rcd

User Module Parameters	Value
FCap	16
ClockPhase	Norm
ASign	Neg
ACap	0
AMux	ACB03
BCap	0
AnalogBus	Disable
CompBus	Disable
AutoZero	On
CCap	0
ARefMux	AGND
FSW1	On
FSW0	On
BSW	On
BMux	ACB02
Power	High

TMR_Knob

User Module Parameters	Value
Clock	VC3
Capture	Row_0_Input_0
TerminalCountOut	None
CompareOut	None
Period	127
CompareValue	0
CompareType	Less Than Or Equal
InterruptType	Compare True
ClockSync	Sync to SysClk
TC_PulseWidth	Full Clock
InvertCapture	Normal

TX

User Module Parameters	Value
Clock	DBB01
Output	Row_0_Output_3
TX Interrupt Mode	TXRegEmpty
ClockSync	Sync to SysClk
Data Clock Out	None

OBSOLETE

Figure 11. Eight Channel Mixer Global Resource Configuration

Global Resources	Value
CPU_Clock	24_MHz (SysClk/1)
32K_Select	Internal
PLL_Mode	Disable
Sleep_Timer	512_Hz
VC1= SysClk/N	6
VC2= VC1/N	16
VC3 Source	VC2
VC3 Divider	11
SysClk Source	Internal
SysClk*2 Disable	Yes
Analog Power	SC On/Ref High
Ref Mux	(Vdd/2)+/-(Vdd/2)
AGndBypass	Enable
Op-Amp Bias	High
A_Buff_Power	High
SwitchModePump	OFF
Trip Voltage [LVD (SMP)]	4.81V (5.00V)
LVDThrottleBack	Disable
Supply Voltage	5.0V
Watchdog Enable	Disable

Name	Port	Select	Drive
inPGA_Rd	P0[0]	AnalogInput	High Z Analog
inPGA_Rc	P0[1]	AnalogInput	High Z Analog
Port_0_2	P0[2]	StdCPU	High Z Analog
Left_out	P0[3]	AnalogOutBuf_0	High Z Analog
Right_out	P0[4]	AnalogOutBuf_2	High Z Analog
mono_out	P0[5]	AnalogOutBuf_1	High Z Analog
inPGA_Ld	P0[6]	AnalogInput	High Z Analog
inPGA_Lc	P0[7]	AnalogInput	High Z Analog
KnobInput0	P1[0]	GlobalInOdd_0	High Z
KnobInput1	P1[1]	GlobalInOdd_1	High Z
KnobInput2	P1[2]	GlobalInOdd_2	High Z
KnobInput3	P1[3]	GlobalInOdd_3	High Z
KnobBank0	P1[4]	StdCPU	Strong
KnobBank1	P1[5]	StdCPU	Strong
KnobBank2	P1[6]	StdCPU	Strong
TX	P1[7]	GlobalOutOdd_7	Strong
RinB	P2[0]	AnalogInput	High Z Analog
LinA	P2[1]	AnalogInput	High Z Analog
RinA	P2[2]	AnalogInput	High Z Analog
LinB	P2[3]	AnalogInput	High Z Analog
Agnd	P2[4]	ExternalAGND	High Z Analog
Port_2_5	P2[5]	StdCPU	High Z Analog
KnobBank3	P2[6]	StdCPU	Strong
KnobEnableButt	P2[7]	StdCPU	High Z

Summary

PSoC 1 has routing schemes that seems to have limitations. But we have seen here, how the switched capacitor blocks and the continuous time blocks can be made to simulate the other and thereby achieve the routing we require.

About the Author

Name: Sampath Selvaraj
Title: Applications Engr Sr Staff
Contact: samp@cypress.com

OBSOLETE

Document History

Document Title: Signal Mixing with PSoC® Switched Capacitor Blocks - AN16833

Document Number: 001-16833

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	2540267	Chris Paiano	07/23/08	New Application Note
*A	2754641	JVY	08/21/09	Changed the document title from “Advanced Internal Analog Routing without Output Buffers” to “Signal Mixing with PSoC Switched Capacitor Blocks”.
*B	3739098	SAMP	09/10/2012	Added Summary. Updated in new template.
*C	4822035	ASRI	07/03/2015	Obsolete document. Completing Sunset Review.

OBSOLETE

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

OBSOLETE

PSoC® is a registered trademark of Cypress Semiconductor Corp. "Programmable System-on-Chip," PSoC Designer is a trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2008-2015. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.