

Using M68HC16 Digital Signal Processing To Build An Audio Frequency Analyzer

By Mark Glenewinkel

INTRODUCTION

This application note demonstrates the use of a microcontroller unit (MCU) with integrated DSP capabilities. The MC68HC16Z1 is a high performance 16-bit MCU that includes on-chip peripheral modules and a CPU module (CPU16). The CPU16 instruction set simplifies the use of digital signal processing algorithms, and makes it easy to implement low-bandwidth filter and control-oriented applications.

OBJECTIVES

The goal of this application note is for an engineer to learn the MC68HC16Z1 well enough to design and build an audio frequency analyzer (AFA). The following intermediate objectives have been defined to help reach this goal.

- Learning the CPU16 instruction set
- Becoming familiar with MC68HC16Z1 modules
- Learning basic MCU I/O hardware and software
- Understanding DSP system concepts with the frequency analyzer
- Understanding and implementing common DSP algorithms with an MCU

This is a tutorial design project that follows a hands-on approach to using DSP. It provides concrete hardware/software applications that are used to understand and design an MCU-based system utilizing DSP algorithms. A basic knowledge of MC68HC16Z1 hardware and the CPU16 instruction set is necessary to complete the design project. A complete discussion of digital signal processing is beyond the scope of this note. However, there are a number of standard textbooks and references available. Please refer to the Motorola publications listed under **REFERENCES** for more information concerning topics and devices discussed in this note.



EQUIPMENT REQUIRED

The following items are needed to build and test the audio frequency analyzer (AFA).

1. An IBM PC compatible computer with a parallel printer port
2. The M68HC16Z1EVB
3. A prototyping or wire-wrap board
4. One straight DB25 cable, male on one end, female on the other
5. A 5 volt power supply
6. An audio sound source, preferably a CD player
7. Two Y-connectors to split the stereo sound source with audio cables
8. A sinusoidal waveform generator, optional
9. Oscilloscope for debugging, optional

All of the components needed to build the AFA are shown in **Figure 4** and **Figure 5**, the AFA schematics.

THE AUDIO FREQUENCY ANALYZER

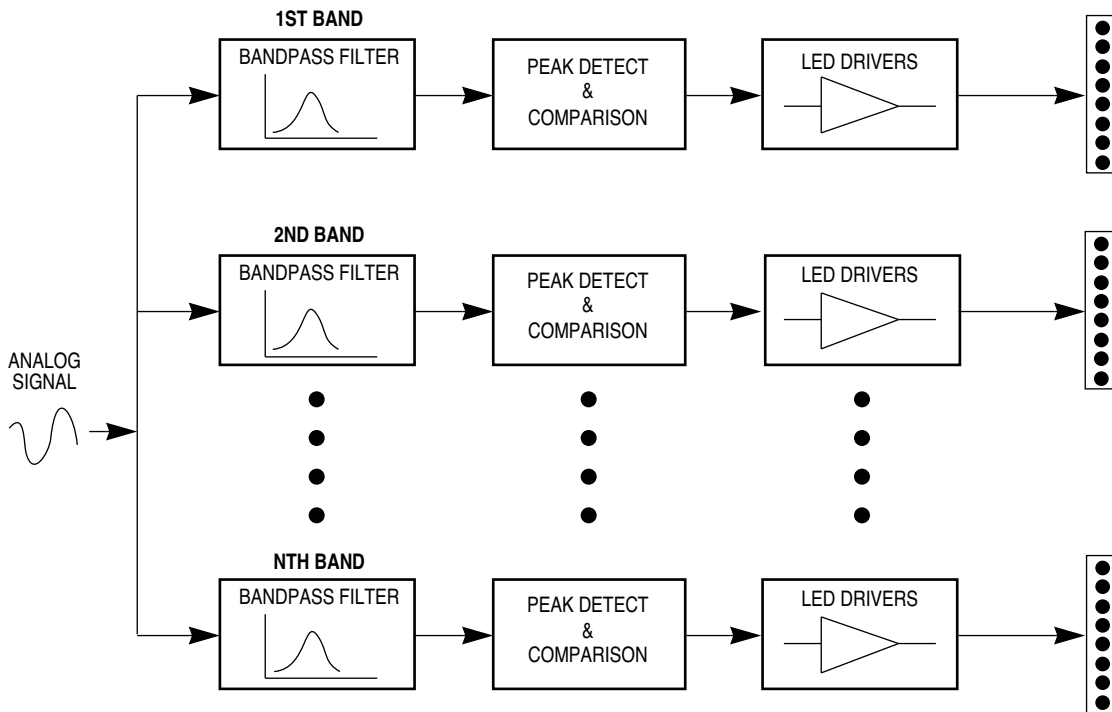
Spectral analysis is a method of determining the specific frequency content of a signal and the energy levels of these frequencies. This information is processed by either Fourier Transform methods or by specific filtering of the signal. The information is tabulated for more analysis or displayed in a visual format.

One example of spectral analysis is found in oil exploration. An engineer sends a known signal into the earth and then calculates the frequency content of the reflected signal. This is a classic input/output black box. The transfer function of the black box (the earth in this case) yields clues to the structure beneath the surface. Different frequency responses correspond to different types of rock. With spectral analysis, the engineer can decide whether it is feasible to drill.

This project focuses on the frequency analysis of an audio signal. A frequency analyzer is often used in audio systems and recording studios. It filters out energy levels of specific audio frequencies and displays them to indicate the frequency content of the audio signal. Audio frequency analyzers are also used in conjunction with equalizers to help the user define and shape the spectral characteristics of a sound source.

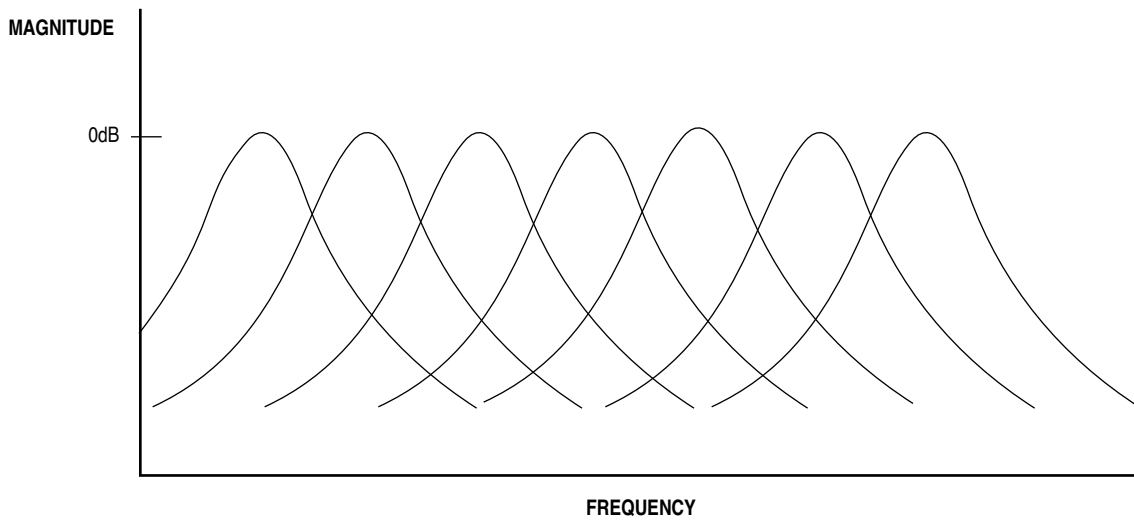
Figure 1 is a generic system diagram of a frequency analyzer based on bandpass filters. The input signal is split and sent to all the filters. The filters pass only specific frequency components of the input signal. After filtration, the strength of each passed signal is analyzed, and the amount of energy in each band is represented on an LED display. This process is executed in a continuous real-time algorithm. **Figure 2** shows a typical audio frequency analyzer transfer function.

Figure 3 is a system diagram of the AFA project, which is implemented using digital filters. Two stereo audio signal inputs are combined by a summing circuit. An anti-aliasing filter removes unwanted high frequency components. A biasing circuit centers the signal around 2.5 vdc for proper analog-to-digital conversion. The ADC module in the MC68HC16Z1 samples the analog signal and digitizes it, then the data is processed by the CPU16. Processing consists of five DSP bandpass filter algorithms. Each determines the amplitude of a specific frequency band and encodes display data. The queued serial peripheral interface (QSPI) is used to send display data to the LED array in real time. Each of these functional blocks is discussed in detail later in this note. Hardware is discussed first, then software.



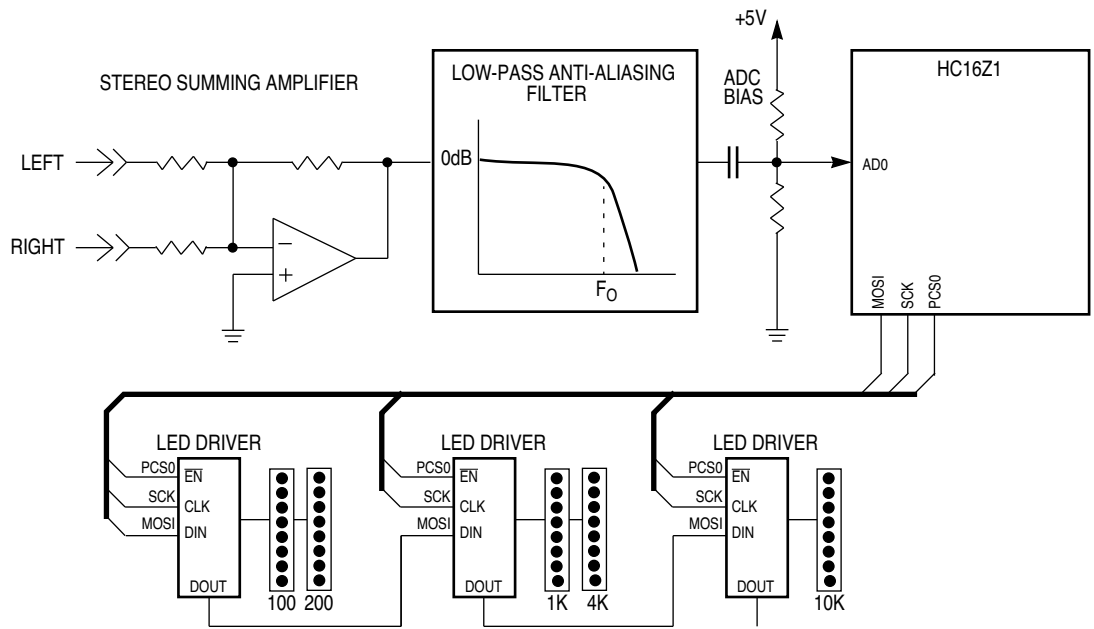
AN1233 F1

Figure 1 Frequency Analyzer System Diagram



AN1233 F2

Figure 2 Bandpass Frequency Analyzer Transfer Function

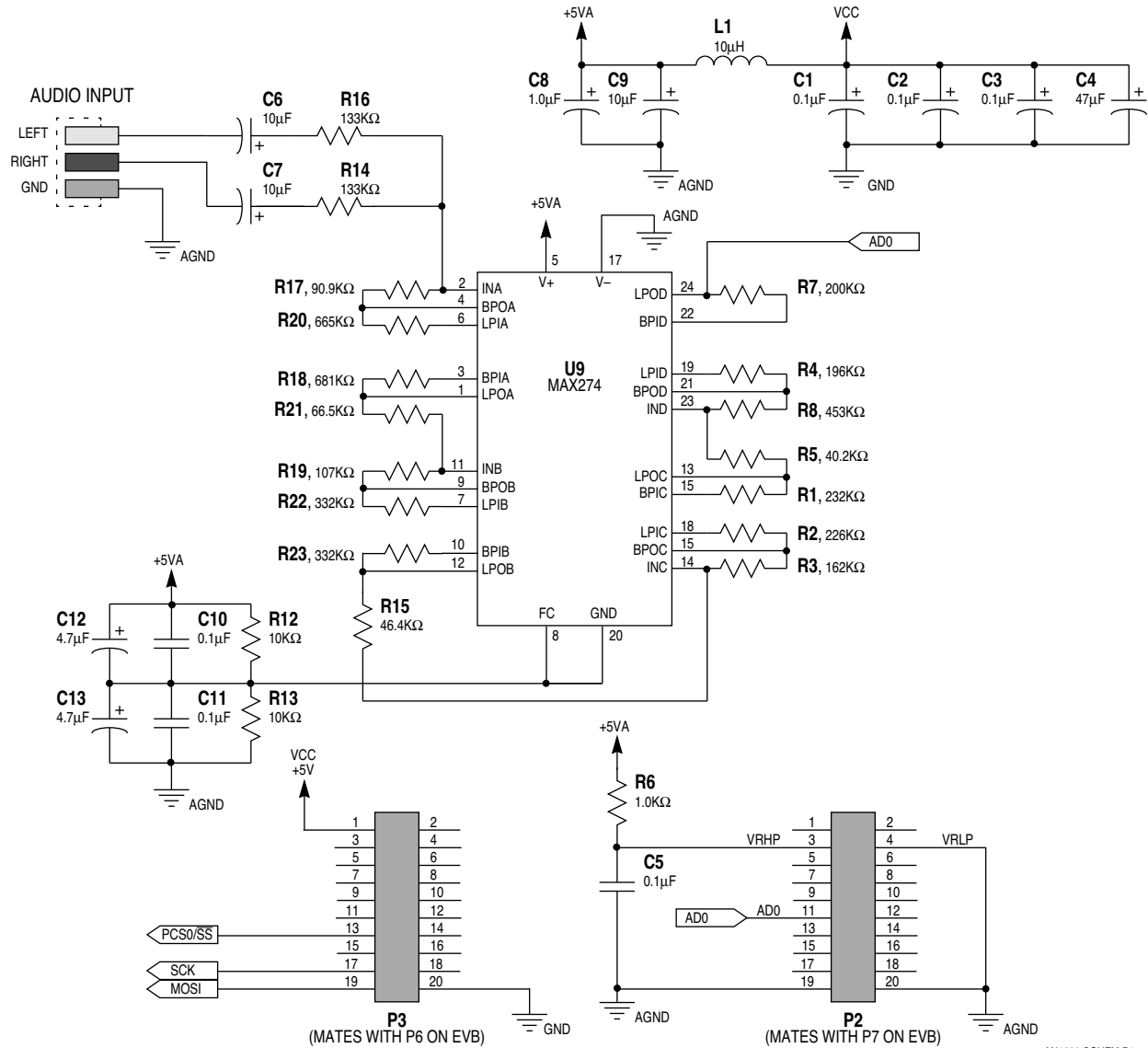


AN1233 F3

Figure 3 Audio Frequency Analyzer System Diagram

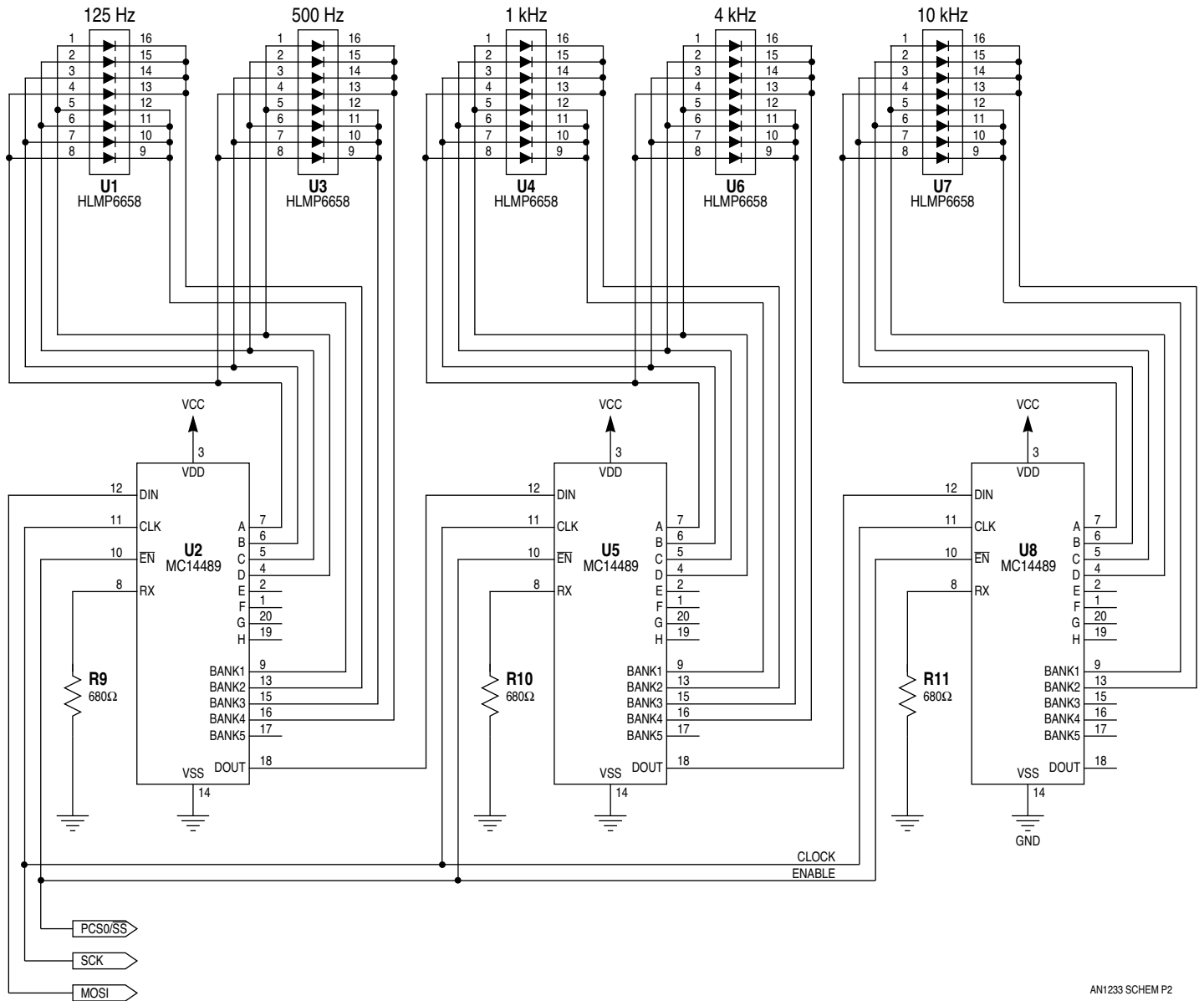
AFA Hardware

Familiarity with the AFA hardware helps to understand the code used to implement the analyzer. **Figure 4** is a schematic of the analog front end of the AFA, and **Figure 5** is a schematic of the display logic.



AN1233 SCHEM P1

Figure 4 AFA Analog Front End



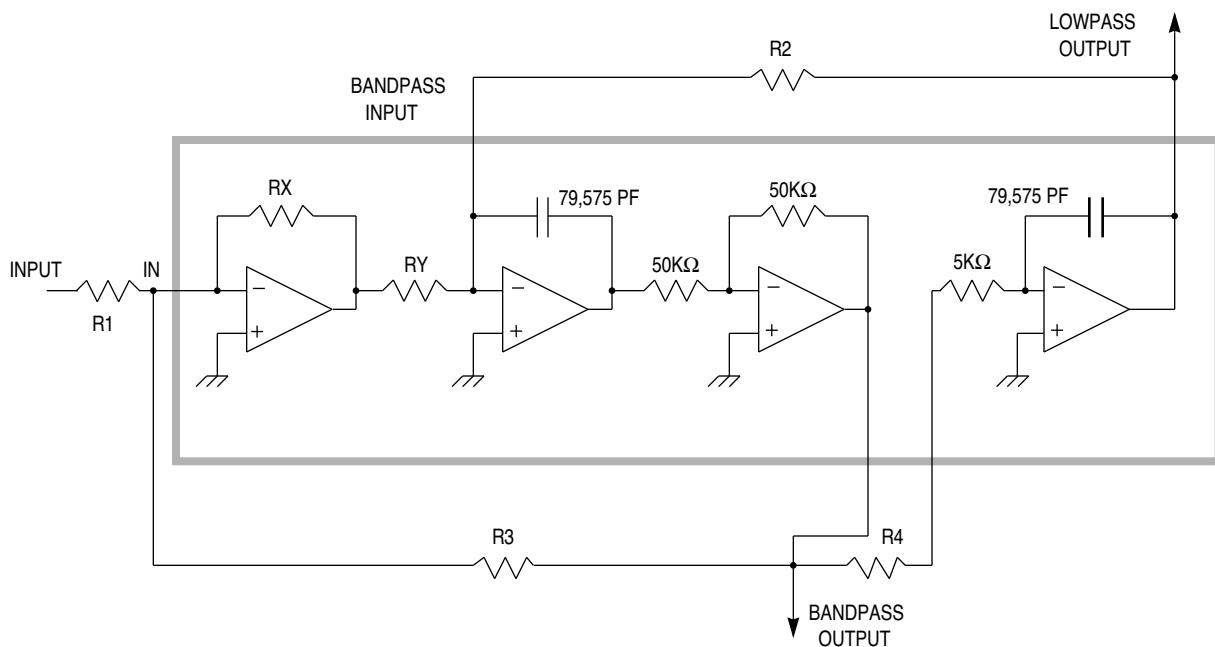
AN1233 SCHEM P2

Figure 5 AFA Digital Back End

The Analog Front End

The analog front end contains all of the circuitry to condition the signal for analog-to-digital conversion and subsequent digital signal processing. It consists of the summing circuitry for the stereo signal, the anti-aliasing filter, and the biasing circuitry for the ADC. A MAX274 low-pass filter chip, manufactured by the Maxim Corporation of Sunnyvale, California, is used to implement all of these functions.

The MAX274 is an eighth order, programmable, continuous-time active filter. The chip consists of four independent cascadable second-order filter sections. Each filter section can implement any all-pole bandpass or lowpass filter, characterized as a Butterworth, Bessel, or Chebyshev response. Each second-order section is programmable with four external resistors. A second-order section is illustrated in **Figure 6**. Maxim provides an evaluation board and a software package that calculates resistor values from response specifications input by the user. This makes the MAX274 very flexible and easy to use when implementing high-order anti-aliasing filters.



AN1233 F6

Figure 6 Second-Order Filter Section

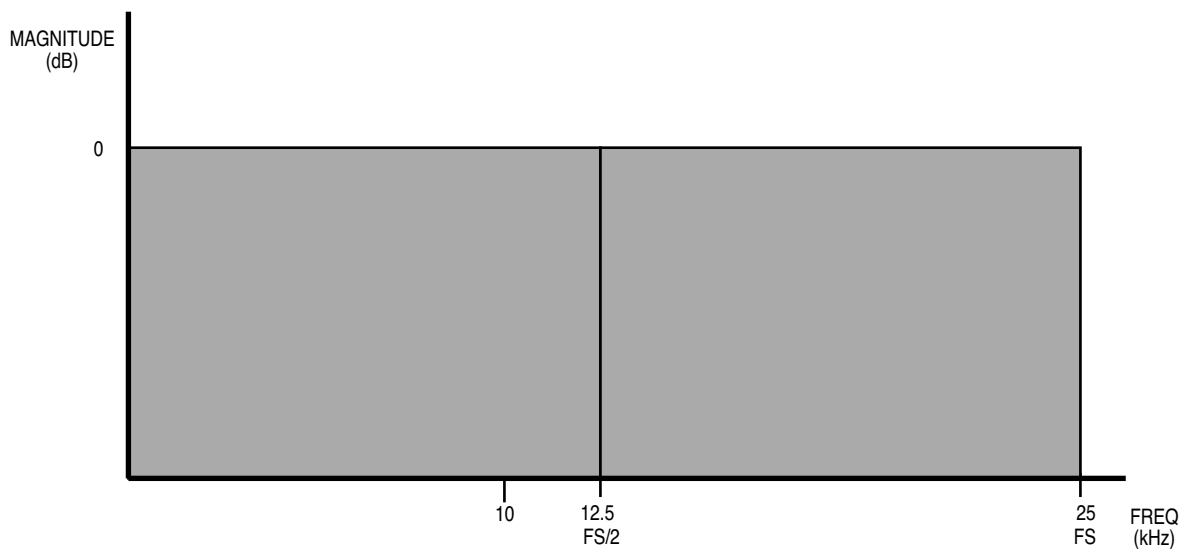
The Summing Amplifier

The summing amplifier combines the two analog stereo signals coming into the system from the audio source. The basic summing circuit shown in **Figure 3** is implemented in the AFA by using an op amp in the first second-order filter section of the MAX274. As shown in **Figure 4**, two summing resistors (R14 and R16) are used to feed the input signals to the inverting input of the op amp, which combines them into one signal.

Anti-Aliasing Filter

When a signal of a given frequency is sampled at too low a rate, it appears as a totally different lower frequency at the output of the sampler. This phenomenon is referred to as aliasing. Aliasing occurs at a point called the folding frequency, which is one-half the sampling frequency. In order for the frequency analyzer to be accurate, sampling frequency must therefore be at least two times the highest frequency component to be sampled. The ideal solution to this problem is to raise the sampling rate as high as possible, but real-world designs generally have a fixed upper limit on sampling frequency. The most practical solution is to attenuate high frequency components of the input signal so that aliasing does not occur. The anti-aliasing filter correctly attenuates the high frequency components of the signal, so that they are not present within the sample bandwidth.

The AFA has a 25-kHz sampling frequency (F_s), and a processing bandwidth of 10 kHz. If no filter is used, signal components with a frequency higher than 12.5 kHz alias at lower frequencies, and the digitized samples represent invalid information. **Figure 7** shows these relationships. $F_s/2$ is the folding frequency, 12.5 kHz. Frequencies that will not alias with a 25 kHz sampling frequency are to the left of $F_s/2$, while frequencies that will alias are to the right of $F_s/2$.



AN1233 F7

Figure 7 AFA Aliasing Without Filter

Anti-aliasing filter design must be a compromise. An efficient and economical solution is to find an intermediate filtration range, between high-order filter roll-off and DSP bandwidth. If the filter has a slow roll-off, a higher sampling frequency is needed, the sampling period is shortened, and there is less time for the DSP algorithm to execute. In other words, a steeper roll-off requires a lower sampling frequency, which in turn provides a longer sampling period for DSP operation.

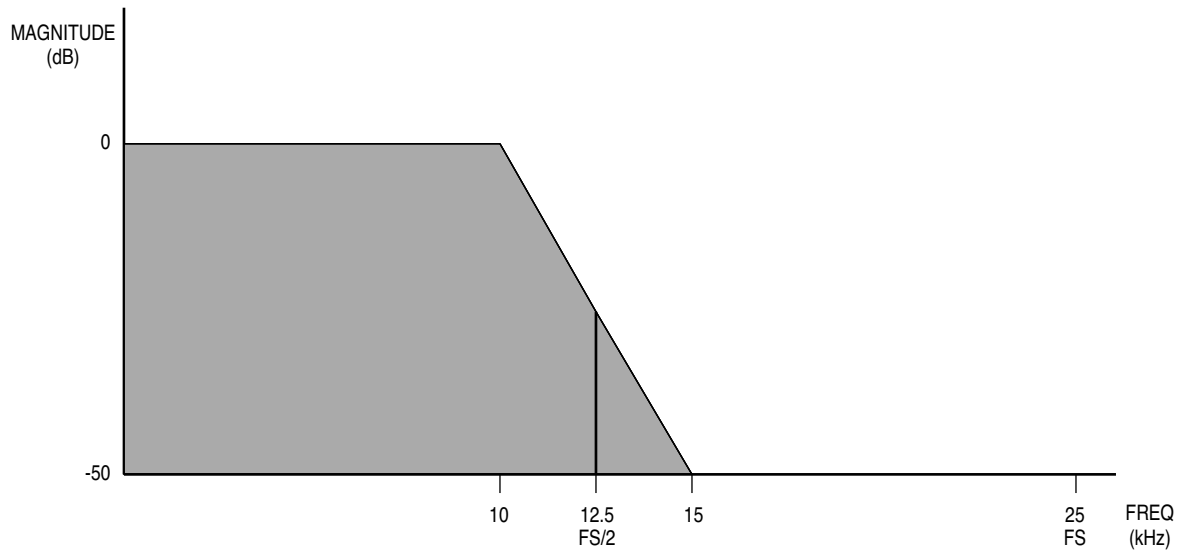
The AFA anti-aliasing filter passes frequencies up to 10 kHz. The filter stop band begins at 15 kHz. Stop band attenuation is dependent upon the application and the dynamic range of the sampled data. In this case, filter output is fed to the ADC module in the MC68HC16Z1, which does not have sufficient resolution or dynamic range to “see” energy in the stop band. An 8-bit conversion that allows a dynamic range of 48 dB is used. The following equation shows these relationships.

$$\text{Voltage Attenuation (dB)} = 20 * \log [1/(2^{\text{ADCres}})]$$

Where:

ADCres = A/D converter resolution

System bandwidth is 10 kHz, and at a 25 kHz sampling frequency, components above 12.5 kHz will alias. Therefore, the signal must be attenuated 48 dB to eliminate all aliasing components. Accordingly, the filter must have a minimum drop-off slope of 96 dB per octave. To insure that this requirement is met, a roll-off of 100 dB per octave is used. Using these values with the MAX274 design software, resistor values for an eighth order 0.5 dB passband ripple Chebyshev filter were obtained. Lower passband ripple was sacrificed to gain steeper roll-off. The anti-aliasing filter response programmed into the MAX274 is shown in **Figure 8**.

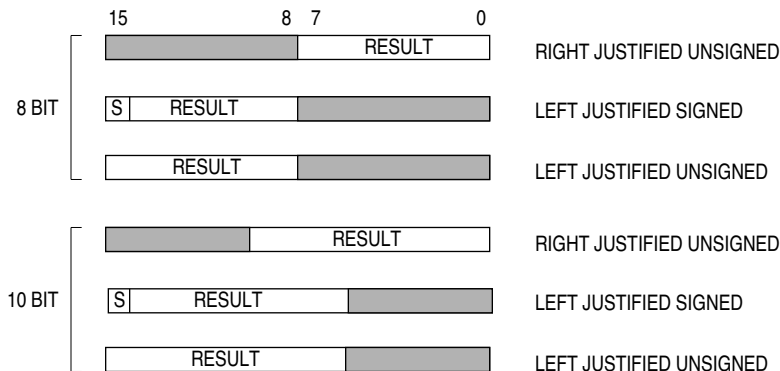


AN1233 F8

Figure 8 AFA Anti-Aliasing Filter Roll-Off

ADC Input Biasing

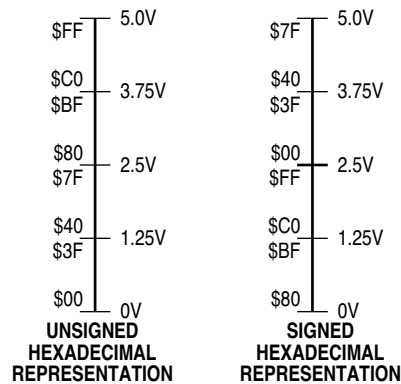
The MC68HC16Z1 ADC module can convert analog data into six different digital representations. Digital data can have 8-bit or 10-bit resolution, can be signed or unsigned, and can be left or right justified. These formats are shown in **Figure 9**.



AN1233 F9

Figure 9 ADC Conversion Formats

Figure 10 shows hexadecimal representations of signed and unsigned ADC data. For 8-bit conversions, there are 256 possible values. Unsigned formats assume the zero voltage point is at the low ADC reference voltage, with 256 steps from low to high reference. Signed formats assume that the zero voltage point is halfway between the low and high ADC reference voltages. The most significant bit indicates a positive or negative value — 128 values represent positive voltages, and 128 two's-complement values represent negative voltages (\$00 represents the midpoint, and \$FF represents midpoint minus one count).



AN1233 F10

Figure 10 Hexadecimal Representation of 8-Bit ADC Data

The AFA uses signed, 8-bit, left justified ADC data. The analog signal must be biased at 2.5 vdc, centered between the 0 vdc and 5 vdc ADC reference voltages, in order to use this representation. The MAX274 is used to bias the signal.

The MAX274 requires two power connections. Biasing circuitry consists of a voltage divider (R12, R13) and decoupling capacitors (C10 – C13) connected to one of the MAX274 supplies. The V– pin is connected to analog ground. The V+ pin is connected to the 5 volt supply. The GND pin is connected to 2.5 volts. This splits the supply and causes the analog signal to have a 2.5 volt DC offset. The signal is buffered by an op amp driver and is sent directly to the ADC module pins from the MAX274. The ADC can now properly sample the signal.

The Digital Back End

The digital back end shown in **Figure 5** contains all of the circuitry required to output digitally processed information to the LED array. When digital signal processing is complete, encoded energy levels for each band are loaded into QSPI transmit RAM, then the QSPI is activated, and the data is transmitted serially to the MC14489 LED drivers.

QSPI software is one of the more difficult aspects of the AFA, but the hardware is quite simple. Three QSPI pins, MOSI, SCK, and PCS0, are used. The master out slave in (MOSI) connection is used to transfer data, the serial clock (SCK) connection is used to clock the transfer, and the peripheral chip select (PCS0) connection is used to enable the LED drivers. The QSPI must be configured correctly to transfer data to the drivers. Refer to the *QSM Reference Manual* (QSMRM/AD) for more information about the QSPI.

The MC14489 LED Driver

The MC14489 can drive individual lamps, seven-segment displays, or combinations of both, in a multiplexed fashion. The chip receives data via a serial input port, and features data retention plus decode and scan circuitry. This reduces software overhead required to perform these tasks. A single current-limiting resistor (Rx) is the only external component needed to operate the MC14489.

Three MC14489 drivers are used in the AFA. There are five 8-bit LED arrays. Two of the MC14489 chips control four banks of four diodes each, and one controls two banks of four diodes each. Drive current for diodes in each bank is supplied by pins A, B, C, and D of the MC14489. The cathodes of each bank of diodes are tied together and a bank-select pin sinks the current for that bank. Please refer to the MC14489 Data Sheet for more information.

The M68HC16Z1 EVB and Development Environment

The M68HC16Z1 Evaluation Board provides the capability to test and debug the audio frequency analyzer. **Table 1** shows development software supplied with the EVB.

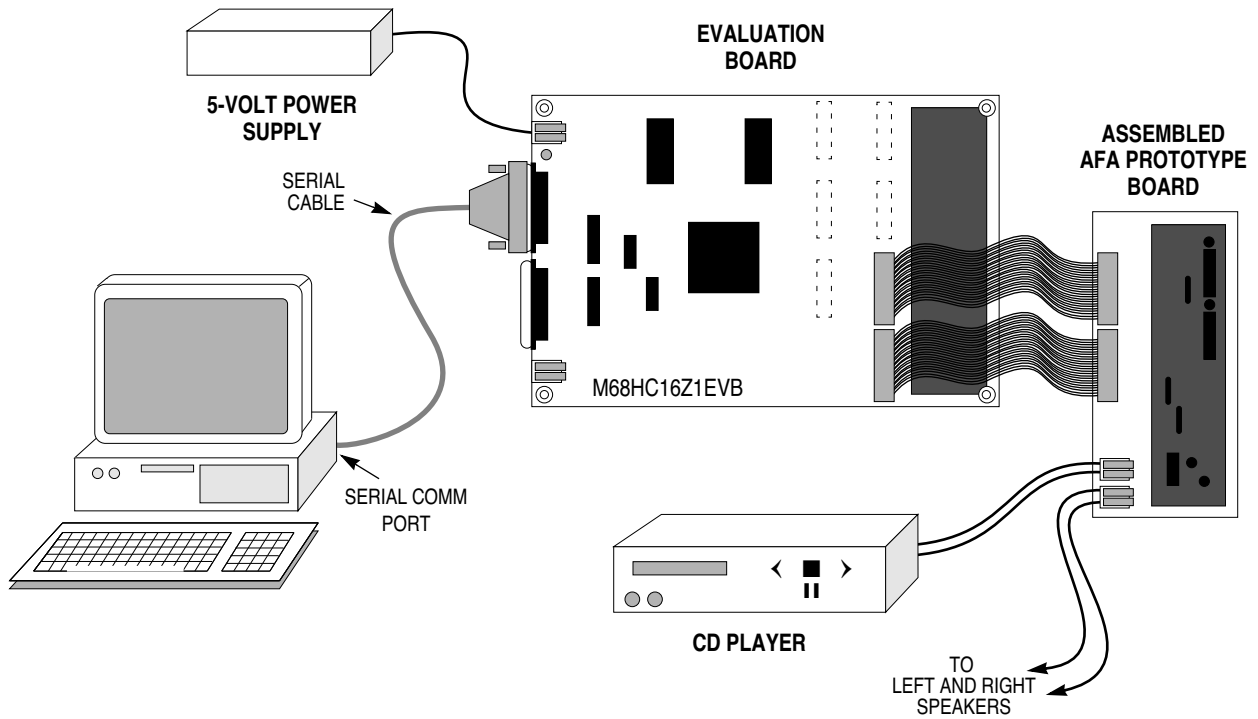
Table 1 Development Software

MASM16.EXE
MASM.EXE
HEX.EXE
MASM16.HLP
EVB16.EXE

MASM16 software is used to edit and assemble code, and EVB16 software is used to download code to the EVB and run it. EVB16 software also has debug capabilities such as trace and breakpoint. Please refer to the *M68HC16Z1EVB User's Manual* for a list of debug features.

Assembling the Development Environment

Assembling the development system with the AFA is simple. Hook up the system as shown in **Figure 11**. The AFA project board connects to the M68HC16Z1EVB via P7 and P6. Use the DB25 cable to connect the parallel port of the PC to the parallel port connector of the EVB. After connecting the 5 volt power supply to the M68HC16Z1EVB, connect the audio signal source. A CD player is the recommended source for a high quality output. Split the audio source outputs so that both the AFA board and the speakers receive the signals (audio splitters can be found at most stereo and electronics stores).



AN1233 F9

Figure 11 AFA Development System Setup

AFA Software

Even though hardware is required to build the AFA, software running on the CPU16 performs most of the actual work. Five tutorial programs must be integrated to complete the project. Each program demonstrates specific functions of the AFA, and each is discussed in a separate section. Since this is a DSP project/tutorial, discussion focuses on signal-processing tasks. Each of the tutorial programs must be modified in order to complete the AFA. The software steps to the AFA design are listed below.

1. Acquisition of data
2. QSPI to MC14489 interface
3. Periodic interrupt timer routine
4. Peak detector
5. 1-kHz bandpass filter routine
6. 5-band audio frequency analyzer

AFA software is listed in **Table 2**. Each of the first six programs in the table corresponds to one of the software steps listed above. In order to organize and streamline the project, each program has been designed according to a standard template for the M68HC16Z1EVB. **Figure 12** shows the template.

Table 2 AFA Project Software

ADC.ASM
QSPI_LED.ASM
INT_TEST.ASM
PEAK.ASM
1K_FLTR.ASM
5BAND_SA.ASM
EQUATES.ASM
ORG00000.ASM
INITSYS.ASM
INITRAM.ASM
OUTVAL1.ASM
OUTVAL2.ASM

OUTVAL1.ASM and OUTVAL2.ASM are lookup tables for the LED display routines. They contain values that correspond to the number of LEDs needed to reflect a given peak value.

In addition, utility files that simplify startup and usage of the MC68HC16Z1 have been included in the AFA software package. A brief description of each include file follows.

EQUATES.ASM provides an equates table of MC68HC16Z1 registers and equivalent address values.

ORG00000.ASM defines the reset vector.

INITSYS.ASM initializes the CPU16, takes care of the extension registers, disables the COP watchdog, and sets system clock speed to 16.78 MHz.

INITRAM.ASM turns on the 1-Kbyte SRAM module, maps the RAM array to address \$10000, and moves the stack pointer to \$103FE to increase interrupt-processing speed.

Source code for all of these files is available on the Motorola Freeware Bulletin Board. The BBS number is (512) 891-3733. The files are archived under the name AFA.ARC, in the AMCU section.

```

*
*   MOTOROLA, INC.
*   Advanced MCU Division
*   Austin, Texas
*
*   Title: HC16 SOFTWARE TEMPLATE
*
*   File Name: TEMPLATE.ASM
*
*   Description: This program provides a template for all
*                designers to use with the HC16Z1
*                An equate table is given.
*                The reset vector is initialized.
*                The CPU and RAM are also initialized.
*                The user can put his code in the 'user area'
*                block of this template
*
*   History: 06/05/91 Created.
*            10/02/91 Modified comments.
*
*   Note: This program is written for the M68HC16Z1EVB.
*****
INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

ORG      $0200          ;start program after interrupt vectors

***** Initialization Routines *****
INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                        ;set sys clock at 16.78 MHz, disable COP
INCLUDE 'INITRAM.ASM' ;initialize and turn on SRAM
                        ;set stack (SK=1, SP=03FE)

***** Start of user program area *****

```

Figure 12 AFA Software Template

Software Design Constraints

It is important to understand the specifications and system constraints on the software. A software flow diagram of the AFA is shown in **Figure 13**. Each of the process boxes in the flowchart corresponds to one of the steps toward the complete design. The main tasks are to convert analog input to digital data, run five infinite impulse response (IIR) bandpass filter routines, detect the peak amplitude of each filtered signal, encode the peak value to an LED display value, update the QSPI transmit RAM, and transmit the information to the LED drivers. The flowchart also shows that the AFA is a real-time digital signal processing algorithm that runs in a continuous loop.

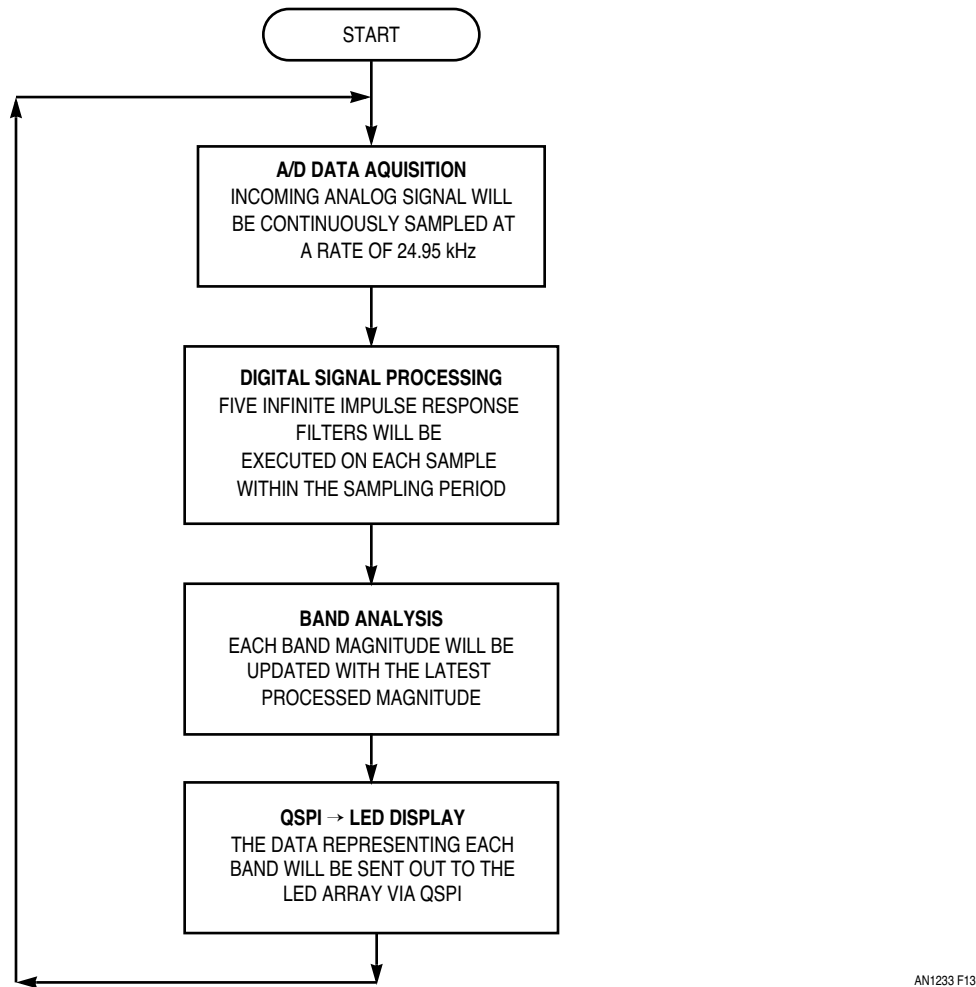


Figure 13 AFA System Software Flowchart

All processing must be completed within one period of the 24.95-kHz sampling frequency. As shown below, a 24.95-kHz sampling frequency is equivalent to a 40.08- μ s sampling period. The MC68HC16Z1 is running at 16.78 MHz, so the system clock period is 60 ns. Thus, all necessary processing must be completed in 668 clock cycles, before the next sampling period begins.

$$F_s = 24.95 \text{ kHz}$$

$$T_s = 1/F_s = 40.08 \text{ } \mu\text{s}$$

$$F_c = 16.78 \text{ MHz}$$

$$T_c = 1/F_c = 60 \text{ ns}$$

$$\text{System clock cycles per sampling period} = T_s/T_c = 668 \text{ system clock cycles}$$

Where:

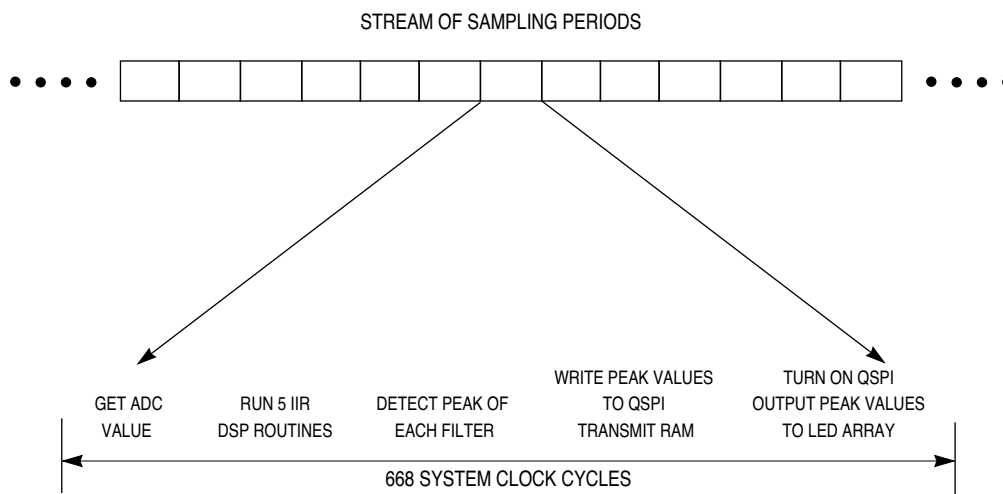
F_s = Sampling frequency

T_s = Sampling period

F_c = MC68HC16Z1 CPU clock frequency

T_c = MC68HC16Z1 CPU clock period

Figure 14 shows the relationship between sampling periods and real-time digital signal processing. All calculations and internal/external housekeeping must be taken care of within the given sample period.



AN1233 F14

Figure 14 AFA Sampling Period

Software Design Implementation

The following sections examine AFA software in detail. For each of the programs, there is a discussion of design and implementation, a code listing, and appropriate flow charts. In the interest of brevity, the standard template headers have been omitted from the listings, and redundant portions of flowcharts are reproduced only once.

Analog-to-Digital Data Acquisition (ADC.ASM)

In order to perform digital signal processing, a digital representation of the analog signal must be available. The MC68HC16Z1 contains a programmable ADC module. The ADC has a number of automatic conversion modes. Only four registers are needed to control the ADC. Refer to the *ADC Reference Manual* (ADCRM/AD) for more detailed information.

ADC.ASM initializes the ADC module, then goes into a continuous loop, repeating the programmed conversion sequence. **Figure 15** is a flowchart of **ADC.ASM**.

To test the routine, first load and assemble the **ADC.ASM** file, then switch to the EVB16 debugger. Download the assembled file to the M68HC16Z1EVB, trace execution until the infinite loop begins to execute, then examine the ADC result registers.

Display the memory locations starting at \$FF710. Check the memory location \$FF711. If the AFA is hooked up properly, a value somewhere between \$74 and \$8B will be displayed. This value is an unsigned representation of 2.5 volts, plus or minus the offset voltage of the MAX274. This same value should also be found at location \$FF730. The signed representation of the same data is found at location \$FF720. The design of **ADC.ASM** is finished. Some of this code will be used to build other programs.

ADC.ASM Code listing

```

INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

ORG    $0200

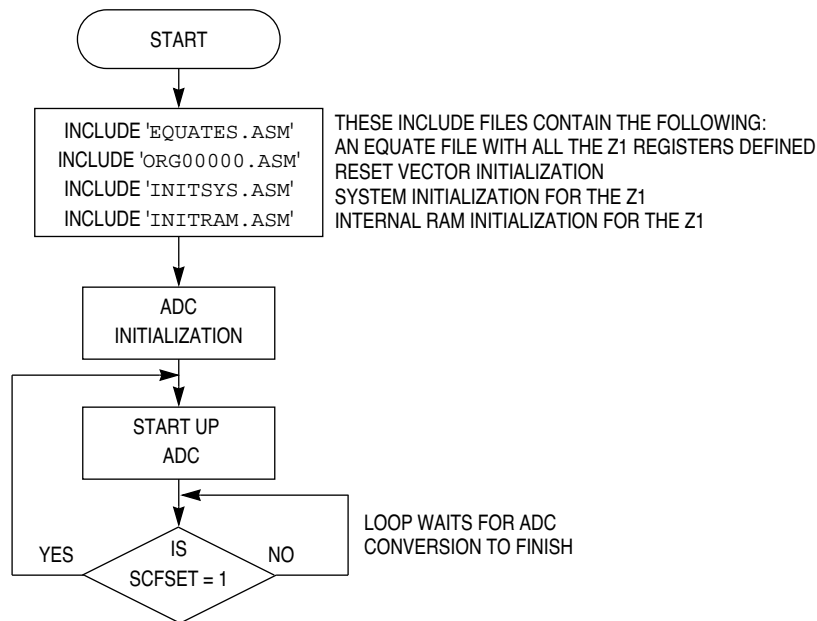
***** Initialization Routines *****

INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                           ;set sys clock at 16.78 MHz, disable COP
INCLUDE 'INITRAM.ASM' ;initialize and turn on SRAM
                           ;set stack (SK=1, SP=03FE)

ORG    $0200
***** ADC Initialization *****
LDD    #$0000
STD    ADCMCR ;turn on ADC
LDD    #$0003
STD    ADCTL0 ;8-bit, set sample period

***** ADC Start *****
LOOP   LDD    #$0000
        STD    ADCTL1 ;single 4 conversion, single channel, AD0
                           ;writing to the ADCTL1 reg starts conversion

        LDAA   #$80
SCFSET BITA ADSTAT ;check for the Sequence Complete Flag
        BEQ   SCFSET ;complete?, if not check again
        BRA   LOOP ;go get another sample
    
```



AN1233 F15

Figure 15 ADC.ASM Flowchart

QSPI TO MC14489 Interface (QSPI_LED.ASM)

This program illustrates QSPI serial timing and data format, which must be understood in order to program the QSPI to talk to the MC14489. The *QSM Reference Manual* (QSMRM/AD) and the MC14489 data sheet are needed to understand the code.

QSPI_LED.ASM initializes the QSPI module and the three MC14489 drivers to handle 40 LEDs. After this it updates the LED array by writing to the MC14489 display registers, then gives control back to the EVB16 development software. Values being sent to the array may be changed either by modifying the memory locations that hold the transmitted data or by reassembling the lines that load these memory locations. **Figure 16** is a flowchart of **QSPI_LED.ASM**.

QSPI_LED.ASM Code Listing

```
INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

ORG      $0200

***** Initialization Routines *****

INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                           ;set sys clock at 16.78 MHz, disable COP
INCLUDE 'INITRAM.ASM' ;initialize and turn on SRAM
                           ;set stack (SK=1, SP=03FE)

***** QSPI Initialization *****
LDAA    #$08
STAA    QPDR ;output pcs0/ss* to 0 when asserted
LDAA    #$0F
STAA    QPAR ;assign QSM port pins to qspi module
LDAA    #$FE
STAA    QDDR ;assign all QSM pins as outputs except miso

LDD     #$8004 ;mstr, womq=cpol=cpha=0
STD     SPCR0 ;16 bits, 2.10MHz serial baud rate
LDD     #$0300 ;no interrupt generated, no wrap mode
STD     SPCR2 ;newqp=0, endqp=3, queued for 4 trans

***** Fill QSPI Command.ram to write the config registers of the 14489
LDAA    #$C0
STAA    CR0 ;cont=1, bitse=1, pcs0=0, no delays needed
STAA    CR1
STAA    CR2
LDAA    #$40
STAA    CR3 ;cont=0, bitse=1, pcs0=0, no delays needed

***** Fill QSPI Transmit.ram to write the config registers of the 14489
LDAA    #$3F
STD     TR0+1 ;store $3F to tran.ram registers
STD     TR2
STD     TR3+1

***** Turn on the QSPI, this will write to the config registers
***** of the MC14489 drivers
GO      LDD     #$8404
        STAA    SPCR1 ;turn on spi
SPIWT   LDAA    SPSR ;after sending data we wait until the
        ANDA    #$80 ;spif bit is set, before we can send more
        CMPA    #$80 ;check for spi done
        BNE     SPIWT

***** Fill QSPI Command.ram to write the display registers of the 14489
LDAA    #$C0
STAA    CR0 ;cont=1, bitse=1, pcs0=0, no delays needed
STAA    CR1
LDAA    #$40 ;cont=0, bitse=1, pcs0=0, no delays needed
STAA    CR2
STAA    CR4
LDAA    #$80 ;cont=1, bitse=0, pcs0=0, no delays needed
STAA    CR3
```

```

***** Fill QSPI Transmit.ram for display registers of the 14489
***** The beginning LED values will be $00, all of the LEDs will be off
LDD    #$8000          ;TR0 = $8000
STD    TR0             ;TR1 = $0080
STAA   TR3+1          ;TR2 = $0000
LDD    #$0080          ;TR3 = $XX80
STD    TR1             ;TR4 = $0000
CLRDL
STD    TR2
STD    TR4

LDD    #$0400          ;display registers need 5 transmissions
STD    SPCR2           ;newqp=0, endqp=4

***** Load up the various LED bands for experimentation
T125   LDAA   #$0F
      STAA   TR4+1      ;125 Hz band
T500   LDAA   #$3F
      STAA   TR4        ;500 Hz band
T1K    LDAA   #$FF
      STAA   TR2+1      ;1k Hz band
T4K    LDAA   #$3F
      STAA   TR2        ;4k Hz band
T10K   LDAA   #$03
      STAA   TR1        ;10k Hz band

LDD    #$8404          ;load up d
STD    SPCR1           ;turn on QSPI

BGND                   ;go back to EVB16 software
                   ;reassemble code for T125 to T10K
                   ;experiment with different values

BRA    T125           ;branch back to TR125 line

```

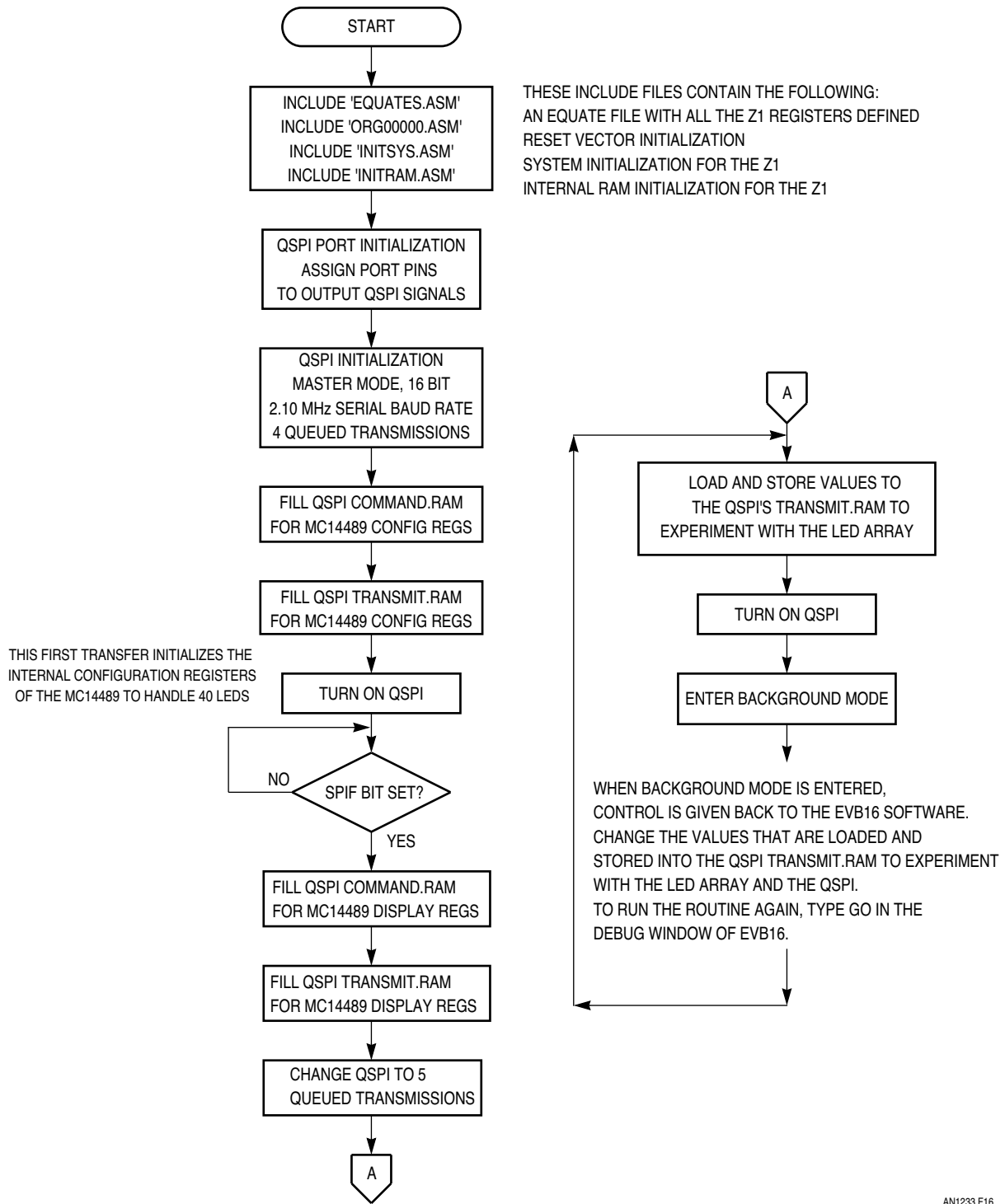


Figure 16 QSPI_LED.ASM Flowchart

AN1233 F16

The Periodic Interrupt Timer (INT_TEST.ASM)

The periodic interrupt timer (PIT) is an internal timer that can be programmed to make an interrupt service request at specific intervals. One application of the PIT is to configure it to interrupt the processor every second so that an interrupt service routine can update a clock.

INT_TEST.ASM produces a square wave on the port F pins of the MC68HC16Z1. The square wave has a set frequency determined by the PIT timeout period. The program uses the level six autovector and the PIT times out at 15.6 ms. Port F is initialized for discrete output, then the code enters a wait loop until the programmed interval elapses. The interrupt service routine creates the square wave. **Figure 17** is a flowchart of **INT_TEST.ASM**.

For detailed information concerning interrupts, the PIT, and port F, refer to the *MC68HC16Z1 User's Manual* (MC68HC16Z1UM/D), the *SIM Reference Manual* (SIMRM/AD), and the *CPU16 Reference Manual* (CPU16RM/AD).

INT_TEST.ASM Code Listing

```
INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

ORG    $0200           ;start program after interrupt vectors

***** Initialization Routines *****

INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                           ;set sys clock at 16.78 MHz, disable COP
INCLUDE 'INITRAM.ASM'  ;initialize and turn on SRAM
                           ;set stack (SK=1, SP=03FE)

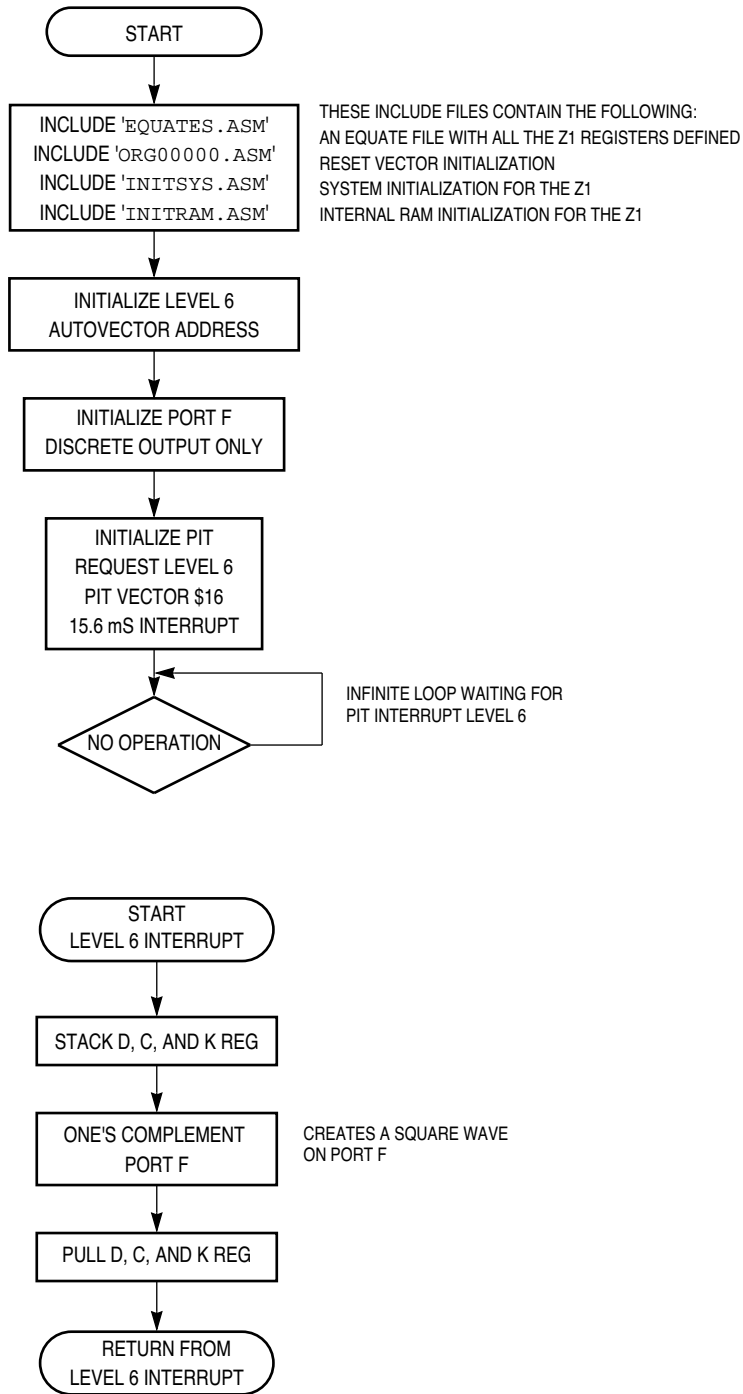
***** Initialize level 6 autovector address
LDAB  #$00
TBEB  ;ek extension pointer = bank0
LDD   #INT_RT          ;load Dacc with interrupt vector addr
STD   $002C           ;store addr to level 6 autovector

***** Initialize PortF *****
LDAB  #$0F
TBEB  ;ek extension pointer = bankf
LDAB  #$00
STAB  PFPAR           ;define port f as discrete i/o
LDAA  #$FF
STAA  DDRF            ;define port f as all output
STAA  PORTF0         ;store $ff to port f

***** Initialize the PIT *****
LDD   #$0616
STD   PICR            ;pirql=6, piv=$16
LDD   #$0080
STD   PITR            ;set the periodic timer at 15.6msec
ANDP  #$FF1F         ;set interrupt priority to 000

***** Infinite loop *****
LOOP  NOP             ;create an infinite loop
      BRA   LOOP      ; waiting for interrupts

***** Exceptions/Interrupts *****
INT_RT PSHM   D,CCR    ;stack Dacc and CCR on stack
      COM   PORTF0    ;one's complement Port F, create square wave
      PULM  D,CCR     ;pull Dacc and CCR from stack
      RTI                    ;return from interrupt
```











AN1233 F17

Figure 17 INT_TEST.ASM Flowchart

Signal Peak Detector (PEAK.ASM)

The signal peak detector graphically measures and displays the peak amplitude of a signal in real time. An audio signal is sampled at 24.95 kHz. The peak amplitude of the signal is detected, then a value that represents the peak on a bar of eight light-emitting diodes (LED) is generated. A reference value of 0.775 Vrms equivalent to 0 dB is used to relate the digital peak value to the LED display. The LED bar can display a signal in the range -15 dB to +6 dB, in 3 dB steps. **Figure 18** shows relationships between the LED bar, decibels, Vrms, and Vp. **Figure 19** shows the relationship between an analog input signal and the peak values displayed. **Figure 20** is a flowchart of **PEAK.ASM**.

LED BAR	dB	VRMS	VPEAK
	+6	1.548	2.187
	+3	1.096	1.548
	0	0.775	1.096
	-3	0.549	0.775
	-6	0.389	0.549
	-9	0.275	0.389
	-12	0.195	0.275
	-15	0.138	0.195

$$dB = 20 \cdot \log\left(\frac{V_{in}}{V_{ref}}\right)$$

$$0dB \geq V_{ref} = 0.775 V_{rms}$$

$$V_{peak} = \sqrt{2} \cdot V_{rms}$$

AN1233 F18

Figure 18 Relationship Between Signal Amplitude and LED Bar

PEAK.ASM code must be downloaded to the EVB in a slightly different manner than usual. The code must be stored in the SRAM array, so it is important to enable and initialize SRAM module correctly before downloading. The steps listed below must be followed when downloading **PEAK.ASM**, **1K_FLTR.ASM**, and **5BAND_SA.ASM**.

1. Download the code
2. Set the IP to \$200
3. Trace the code until you have executed the section labeled 'RAM and Stack Initialization'
4. Set the IP to \$200
5. Download the program again.

The code originating in the internal RAM will now be correctly loaded into the MC68HC16Z1.

PEAK.ASM reads values from a look-up table in memory. The file **OUTVAL2.ASM** contains the table. Be sure this file is in the same directory as **PEAK.ASM** before assembly.

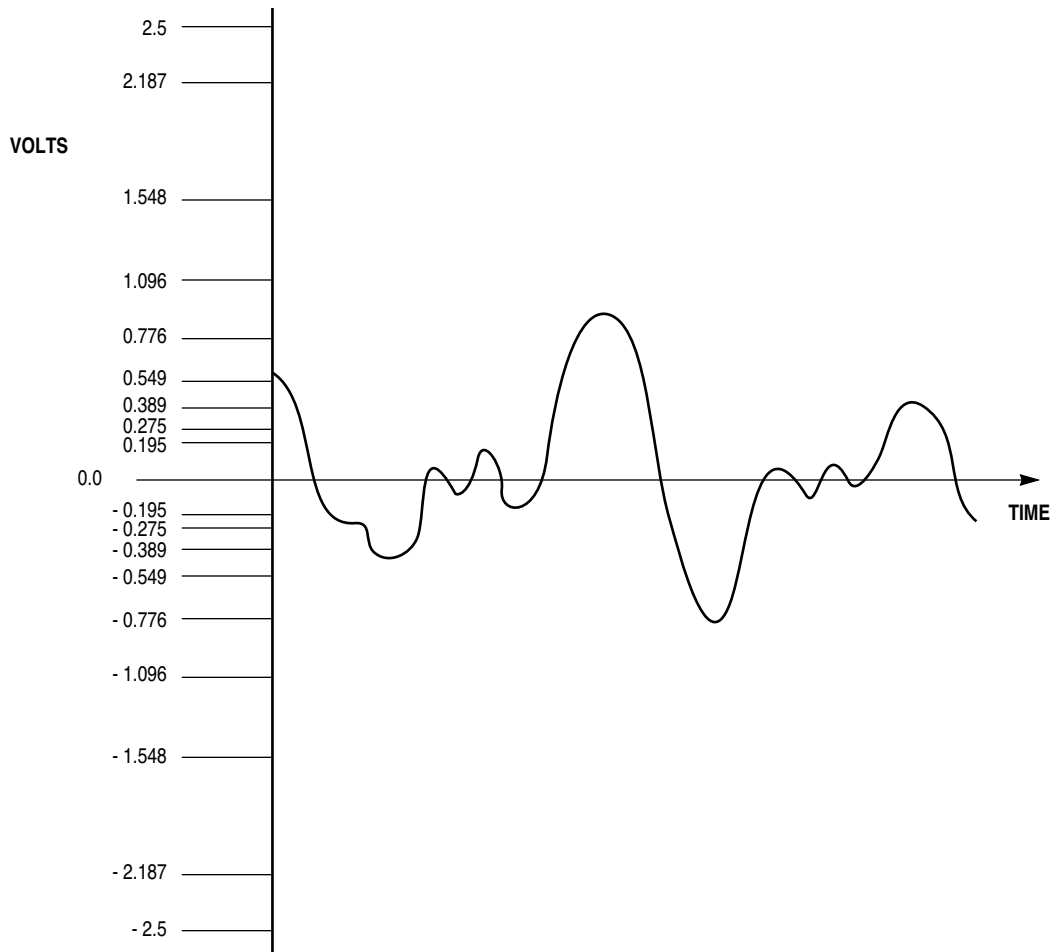


Figure 19 Analog Input vs Peak Display Level

After initializing the SRAM, the ADC, the QSPI, and the PIT, the code jumps to internal RAM at location \$F0000. Internal RAM access time is less than access time for the external RAM on the EVB. This extra speed is important to subsequent programs that use DSP routines.

The program then loops continuously, reading the ADC, encoding the ADC value to its equivalent LED value, and checking to see if the current value is greater than the previous peak value. If so, then the current peak value is updated and stored away in memory. The code can only increase the current peak value.

Peak value encoding is accomplished by self-modifying code that reads values from a look-up table in memory. Cycle counts for each instruction are given on the right-hand side comment line in **OUTVAL2.ASM**. They are used to determine the delay that is needed to create the 24.95-kHz sampling frequency.

The LED array is updated with the current peak value every 10.26 ms. This routine only detects and displays increases of the peak value. In order to follow a changing signal, the peak value must also be decreased periodically. A PIT interrupt performs this task every 62.5 ms. When the PIT times out, the interrupt service routine decrements the peak value and sends a new value to the display.

Using a PIT interrupt to decrement the peak value causes the LED display to decrease slowly, like a capacitor discharging, when the input signal decreases rapidly. This gives the display a more fluid appearance when rapidly-changing peak values are measured. If the display jumped from peak to peak, the discontinuity would lower the aesthetic appeal. In fact, most commercial audio analyzers show the relative peak differences of the frequency spectrum rather than attempt to display the peak signal precisely.

To test the code, hook up the system as shown in **Figure 11**. Input a known signal and observe the display. Apply an audio signal from the sound source and watch the peak detector execute in real time. If there is only one sound source output, connect it to either the left or right AFA input. The display is calibrated to the output of a CD player. The CD player puts out a line level signal, with .775 Vrms equal to 0 dB. If the sound source is not a CD player, adjust the output of the sound source so that the dynamic range of the signal is fully displayed.

PEAK.ASM Code Listing

```

INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

**** Temporary variable storage
PK EQU $0200 ; bank F
CNT EQU $0201 ; bank F

ORG $0200

**** Initialization Routines ****

INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
;set sys clock at 16.78 MHz, disable COP

**** RAM and Stack Initialization
LDD #$00FF
STD RAMBAH ;store high ram array, bank F
LDD #$0000
STD RAMBAL ;store low ram array, 0000
CLR RAMMCR ;enable ram
LDAB #$0F
TBSK ;set SK to bank F for system stack
LDS #$02FE ;put SP in 1k internal SRAM

**** Initialize level 6 autovector address
LDAB #$00
TBK ;ek extension pointer = bank0
LDD #JMPINT ;load Dacc with interrupt vector addr
STD $002C ;store addr to level 6 autovector

**** Initialize the PIT ****
LDAB #$0F
TBK ;ek extension pointer = bankf
LDD #$0616
STD PICR ;pirql=6, piv=$16
LDD #$0101
STD PISR ;set the periodic timer at 62.5msec
ANDP #FFF1F ;set interrupt priority to 000

**** QSPI Initialization ****
LDAA #$08
STAA QPDR ;output pcs0/ss* to 0 when asserted
LDAA #$0F
STAA QPAR ;assign QSM port pins to qspi module
LDAA #$FE
STAA QDDR ;assign all QSM pins as outputs except miso

LDD #$8004 ;mstr, womq=cpol=cpha=0
STD SPCR0 ;16 bits, 2.10MHz serial baud rate
LDD #$0300 ;no interrupt generated, no wrap mode
STD SPCR2 ;newqp=0, endqp=3, queued for 4 trans

**** Fill QSPI Command.ram to write the config registers of the 14489
LDAA #$C0
STAA CR0 ;cont=1, bitse=1, pcs0=0, no delays needed

```

```

        STAA    CR1
        STAA    CR2
        LDAA    #$40
        STAA    CR3                ;cont=0, bitse=1, pcs0=0, no delays needed

*****  Fill QSPI Transmit.ram to write the config registers of the 14489
        LDAA    #$3F
        STD     TR0+1                ;store $3F to tran.ram registers
        STD     TR2
        STD     TR3+1
*****  Turn on the QSPI, this will write to the config registers
*****  of the MC14489 drivers
GO      LDD     #$8404
        STAA    SPCR1                ;turn on spi
SPIWT   LDAA    SPSR                ;after sending data we wait until the
        ANDA    #$80                ;spif bit is set, before we can send more
        CMPA    #$80                ;check for spi done
        BNE     SPIWT

*****  Fill QSPI Command.ram to write the display registers of the 14489
        LDAA    #$C0
        STAA    CR0                ;cont=1, bitse=1, pcs0=0, no delays needed
        STAA    CR1
        LDAA    #$40                ;cont=0, bitse=1, pcs0=0, no delays needed
        STAA    CR2
        STAA    CR4
        LDAA    #$80                ;cont=1, bitse=0, pcs0=0, no delays needed
        STAA    CR3

*****  Fill QSPI Transmit.ram for display registers of the 14489
*****  The beginning LED values will be $00, all of the LEDs will be off
        LDD     #$8000
        STD     TR0                ;TR0 = $8000
        STAA    TR3+1                ;TR1 = $0080
        LDD     #$0080
        STD     TR1                ;TR2 = $0000
        CLRD    ;TR3 = $XX80
        STD     TR2                ;TR4 = $0000
        STD     TR4

        LDD     #$0400                ;display registers need 5 transmissions
        STD     SPCR2                ;newqp=0, endqp=4

*****  ADC Initialization          *****
        LDD     #$0000
        STD     ADCMCR                ;turn on ADC
        LDD     #$0003
        STD     ADCTL0                ;8-bit, set sample period

*****  Initialize the extension registers for the internal ram in bank F
*****  Set up the extension registers to point to bank F
        LDAB    #$0F                ;load b with $0F
        TBEX                    ;transfer Bacc to Ek
        TBXK                    ;transfer Bacc to Xk
        TBYK                    ;transfer Bacc to Yk
        TBZK                    ;transfer Bacc to Zk
        JMP     RAM                ;jump to internal ram for speed!

*****  Start of Internal 1K RAM
        ORG     $F0000
RAM     CLR     CNT                ;clear LED update counter
        CLR     PK                ;clear peak value

LP      CLRD                    ; 2 clear Dacc
        STD     ADCTL1                ; 6 single 4 conversion, single channel AD0
        ; writing to the ADCTL1 reg starts conv
        LDE     LJSRR0                ; 6 load e with x(n), left jus adc result0

*      Check if LEDs need updating
        LDAA    CNT                ; 6 load Aacc with count
        ADDA    #1                ; 2 add 1 to Aacc
        STAA    CNT                ; 6 store new count
        BNE     TRAN                ; 6,2 check to see if its time to update
        ; the LEDs, time = 256 * 668 cycles
        ; 668 cycles = 40.08usec
        ; so LED update time is 10.26msec

```

```

LDD    #$8404          ; 6 load up d
STD    SPCR1           ; 6 turn on QSPI, send LED data out

*      Get LED encode value from look-up table
TRAN   TED             ; 2 transfer Eacc to Dacc
STAA   LD+3           ; 6 Dacc high byte -> instruction ldaa $03??
NOP    ; 2 no operation, wait for CPU pipeline
NOP    ; 2 no operation, wait for CPU pipeline
LD     LDAA    LED_TBL ; 6 load Aacc with the encoded LED value
      ; from scaled peak LED table

*      Update peak value if needed
CMPA   PK             ; 6 compare value to previous peak value
BLS    DN             ; 6,2 branch if not more than peak value
STAA   PK             ; 6 store new peak value
STAA   TR1            ; 6 store new value to all 5 qspi tran.rams
STAA   TR2            ; 6
STAA   TR2+1          ; 6
STAA   TR4            ; 6
STAA   TR4+1          ; 6

***** Loop to generate calculated delay
***** Clocks = 6 + 8*(N-1) N >= 1
***** N is the number put into the B accumulator

DN     LDAB    #$4B    ; 75dec this loop will create an extra delay
WAIT   DECB    ; to make a 24.95kHz sampling rate
      BNE     WAIT    ; or a 668 cycle sampling period
      ; 598 cycles

      JMP     LP      ; 6 jump back to start another conversion

***** Exceptions/Interrupts *****
***** This interrupt is used to decrement each LED bar value
***** representing the peak value of the audio signal
INT_RT PSHM    D,CCR    ;stack Dacc and CCR on stack
      LDAA   PK        ;load Aacc with peak value
      BEQ   DONE       ;equal to 0?, then done

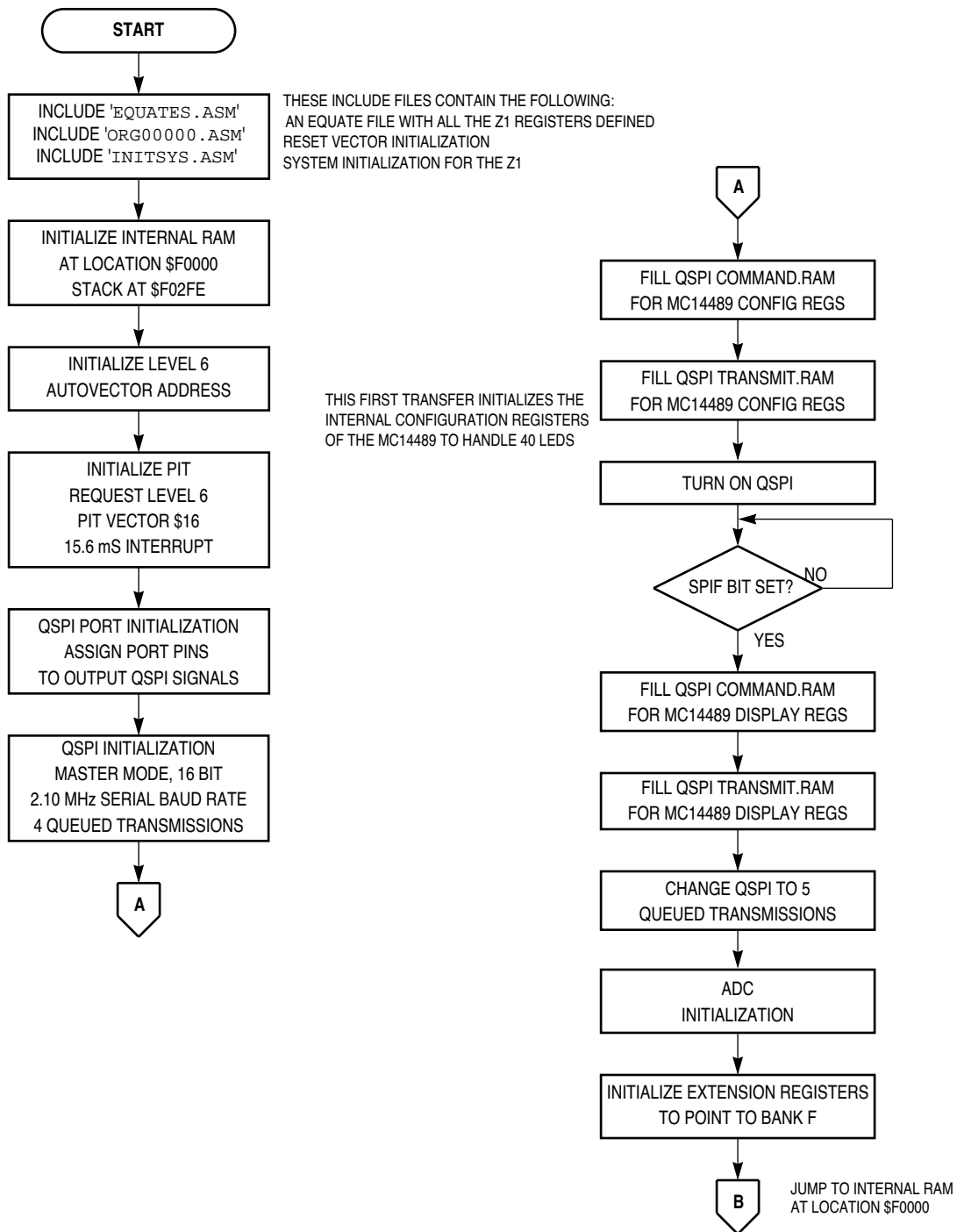
      ANDP   #$FEFF    ;clear C bit
      RORA  ;rotate right once, decrease peak value
      STAA  TR1        ;store Aacc to all qspi tran.ram
      STAA  TR2
      STAA  TR2+1
      STAA  TR4
      STAA  TR4+1
      STAA  PK        ;store Aacc to peak value
      LDD   #$8404    ;load up Dacc
      STD   SPCR1     ;turn on QSPI, send LED data out

DONE   PULM    D,CCR    ;pull Dacc and CCR from stack
      RTI    ;return from interrupt

***** Location of start of level 6 interrupt, has to be in bank 0
*****
JMPINT JMP     INT_RT

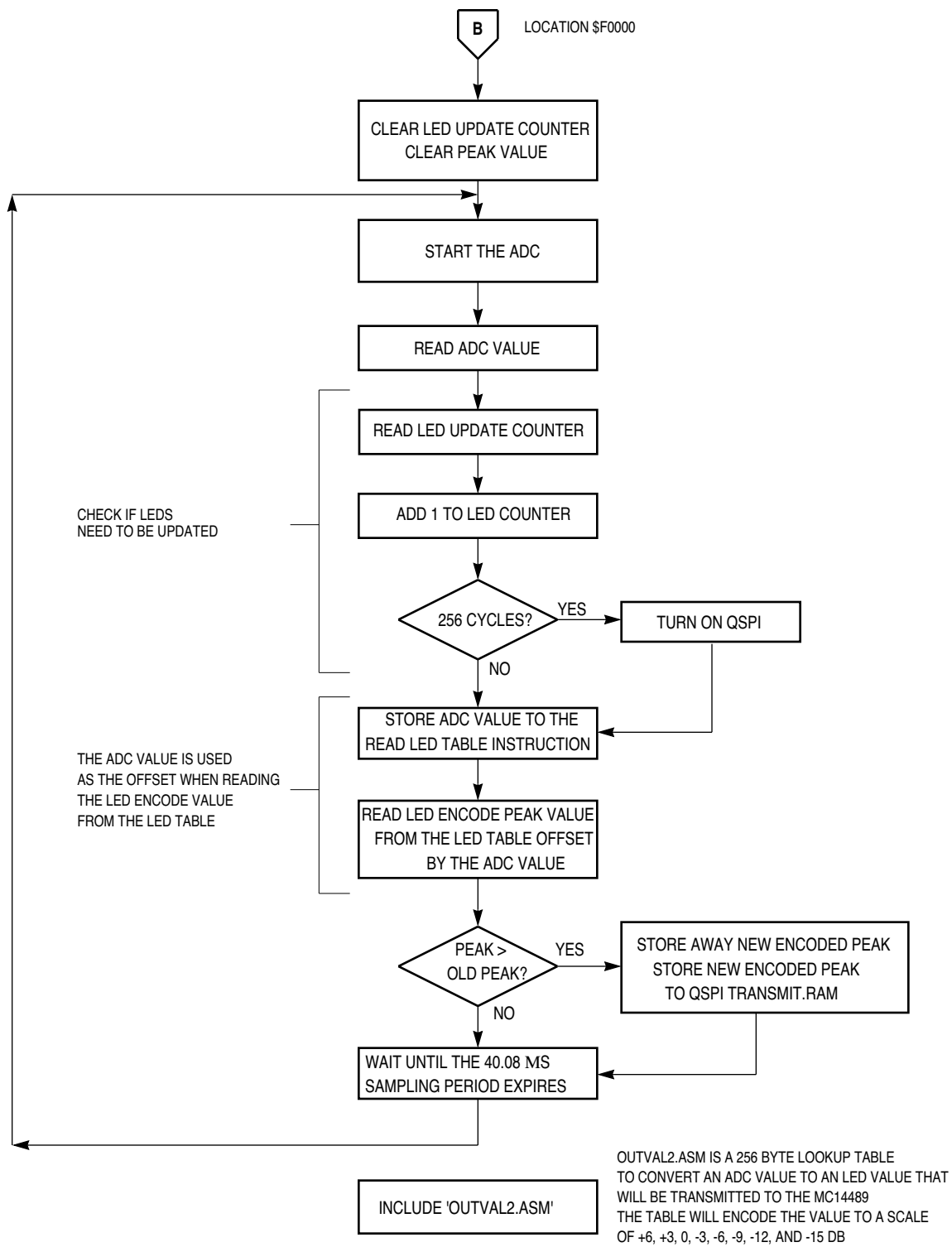
***** OUTVAL2 is a 256 byte lookup table to convert an
***** ADC reading to a LED value that can be transmitted to the 14489
***** Encodes to a scale of +6, +3, 0, -3, -6, -9, -12, -15 dB
      INCLUDE 'OUTVAL2.ASM' ;LED Look up table

```



AN1233 F20A

Figure 20 PEAK.ASM Flowchart (Sheet 1 of 2)



AN1233 F20B

Figure 20 PEAK.ASM Flowchart (Sheet 2 of 2)

A 1-kHz Bandpass Filter (1K_FLTR.ASM)

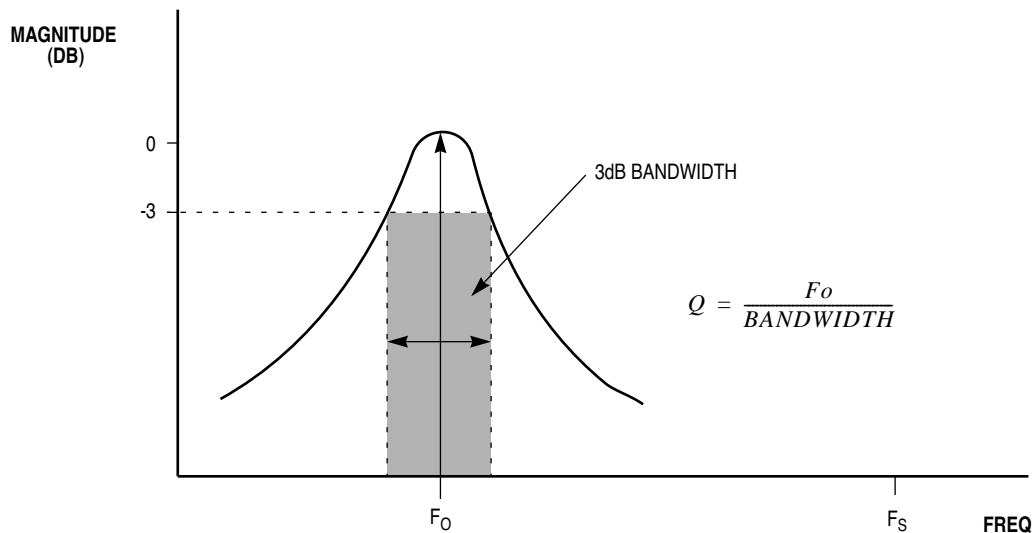
This code is similar in function to the peak detector, except that it executes a 1-kHz IIR bandpass filter on the input signal. The peak is detected and displayed on an LED bar in real time. The focus is on using the MC68HC16Z1 to implement the digital filter. **Figure 22** is a flowchart of **1K_FLTR.ASM**.

The objective is to take incoming sampled data $x(n)$, and run the bandpass filter function on the sample to produce output $y(n)$. Again, this is the basic 'black box' concept of electrical engineering — excite the input and watch the output change. The function in the 'black box' is defined below.

$$y(n) = 2 * \{ \alpha * [x(n) - x(n-2)] + \gamma * y(n-1) - \beta * y(n-2) \}$$

This function implements an IIR bandpass function with characteristics defined by the coefficients α , β , and γ . In an RLC bandpass filter circuit, resistors, capacitors, and inductors would characterize filter response. In the digital implementation of the filter, the α , β , and γ coefficients determine the response in much the same way.

The basic parameters that define digital filter response are the Q , the sampling frequency (F_s), and the center frequency (F_o). The Q value defines the sharpness of the filter and is equal to the center frequency divided by the bandwidth between the 3 dB points. The specified sampling frequency is 24.95 kHz, the center frequency is 1 kHz, and Q value is 1.5. **Figure 21** illustrates these relationships. **Table 3** shows the way in which coefficients are stored in memory.



AN1233 F21

Figure 21 Filter Relationships

Table 3 DSP Filter Algorithm Memory Use

XN1_1K	$x(n-1)$	YN1_1K	$y(n-1)$	GAM_1K	γ
XN2_1K	$x(n-2)$	YN2_1K	$y(n-2)$	BET_1K	$-\beta^1$
		X_2_1K	$x(n) - x(n-2)$	ALP_1K	α

1. To speed processing, the calculated β coefficient is made negative, then added to the expression, rather than subtracted as shown in the equation above.

Equations that define the coefficients are shown below. Coefficient values are also given in the code listing.

$$\begin{aligned}\theta &= \{(2 * \pi * F_0) / F_s\} \\ X &= \theta / (2 * Q) \\ \text{If } X > \pi / 4 &\text{ then } X = 0.75398 \\ \beta &= 0.5 * \{1 - \tan(X)\} / \{1 + \tan(X)\} \\ \gamma &= (0.5 + \beta) * \cos \theta \\ \alpha &= (0.5 - \beta) / 2\end{aligned}$$

Where:

$$\begin{aligned}F_0 &= 1 \text{ kHz} \\ F_s &= 24.95 \text{ kHz} \\ Q &= 1.5\end{aligned}$$

For more information concerning these equations, refer to Motorola Application Note *Digital Stereo 10-Band Generator* (APR2/D).

Once coefficient values have been obtained, they must be encoded. The assembler does not understand fractional decimal numbers, so fractional values are converted into signed 16-bit hexadecimal values. When using two's complement arithmetic, the most significant bit (bit 15) is the sign bit, and the fraction is contained in bits 14 to 0. Fifteen bits can represent the decimal numbers from 0 to 32,767. Multiply the decimal fraction by 32,768, then convert the value to the hexadecimal equivalent. Make certain that hexadecimal equivalents of negative values are in two's complement form. An example is given below.

$$\text{Decimal fraction} = 0.5$$

Multiply fractional decimal value by 32,768

$$0.5 * 32,768 = 16,384$$

Change decimal value to hexadecimal and binary values

$$16,384 \text{ dec} = 4000 \text{ hex} = 0100 \ 0000 \ 0000 \ 0000 \ \text{bin}$$

4000 hex is the 16-bit fractional value.

CPU16 multiply and add instructions are used to implement the function. Processing is streamlined so that, in the final AFA design, five filters can be implemented in the 40.08 μ s sampling period. For a more thorough discussion of the DSP instruction set and related CPU16 architecture, please consult Chapter 11 in the *CPU16 Reference Manual* (CPU16RM/AD). The processing sequence is as follows.

The ADC value $x(n)$ is divided by two to prevent overflow.

The subtraction operation, $x(n) - x(n - 2)$ is performed and the result is stored in location X_2_1K.

Three MAC instructions are executed, starting at address YN1_1K.

The value $y(n - 1)$ is multiplied by γ and added to the M accumulator.

The value $y(n - 2)$ is multiplied by $-\beta$ and added to the M accumulator.

The value $[x(n) - x(n - 2)]$ is multiplied by α and added to the M accumulator.

The M accumulator is multiplied by two, using a left shift instruction, to obtain the $y(n)$ value.

The x and y terms are updated before the next sample is processed:

$$x(n - 1) \text{ becomes } x(n - 2) \text{ and } x(n) \text{ becomes } x(n - 1)$$

$$y(n - 1) \text{ becomes } y(n - 2) \text{ and } y(n) \text{ becomes } y(n - 1)$$

As mentioned earlier, the 1-kHz bandpass filter is very similar to the peak detector design. Once the DSP is finished on the input $x(n)$ sample, the peak detect algorithm is executed.

The include file **OUTVAL1.ASM** is used to encode the DSP output with an LED display value multiplied by two. Be sure this file is in the same directory as 1K_FLTR.ASM during assembly.

The best way to test this program is to connect a signal generator with sine-wave sweep capability to the AFA inputs, then set it to sweep from 0 to 15 kHz. The 1-kHz LED bar should display the amplitude of a pure 1-kHz tone and the routine should filter out higher and lower frequency signals. Since Q is equal to 1.5, some side-lobe frequencies in the pass band should be evident. For instance, if a 2-kHz pure signal is sent into the filter, the side-lobe response of the 1-kHz bandpass will pass an attenuated level of the 2-kHz tone.

1K_FLTR.ASM Code Listing

```

        INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
        INCLUDE 'ORG00000.ASM' ;initialize reset vector

*****  Addresses of coefficients for the IIR Filters and initialization
COEFBS EQU    $0280          ;base addr of coefficients
GAM_1K EQU    COEFBS+$0      ;addr of the gamma coef
BETA_1K EQU    COEFBS+$2     ;addr of the beta coef
ALPH_1K EQU    COEFBS+$4     ;addr of the alpha coef
        ORG    $F0280
        dc.w  $7257          ;1k Hz gamma coef, Q=1.5
        dc.w  $C9F0          ;1k Hz beta coef, Q=1.5
        dc.w  $04F7          ;1k Hz alpha coef, Q=1.5

*****  Addresses of filter terms for the x(n) terms and initialization
XTRMBS EQU    $02A0          ;base addr of x(n) filter terms
XN1_1K EQU    XTRMBS+$0      ;x(n-1)
XN2_1K EQU    XTRMBS+$2     ;x(n-2)
        ORG    $F02A0
        dc.w  $0000          ;1k Hz x(n-1)
        dc.w  $0000          ;1k Hz x(n-2)

*****  Addresses of filter terms for the y(n) terms and initialization
YTRMBS EQU    $02C0          ;base addr of y(n) filter terms
YN1_1K EQU    YTRMBS+$0      ;y(n-1)
YN2_1K EQU    YTRMBS+$2     ;y(n-2)
X_2_1K EQU    YTRMBS+$4     ;x(n) - x(n-2), stored here for mac
        ORG    $F02C0
        dc.w  $0000          ;1k y(n-1)
        dc.w  $0000          ;1k y(n-2)
        dc.w  $0000          ;1k [ x(n) - x(n-2) ]

*****  Addresses of various temporary variables and initialization
PKRES EQU    $02E0          ;base addr of filter result storage
PK_1K EQU    PKRES+$0        ;peak value for 1k Hz
CNT EQU    PKRES+$1         ;count value for LED qspi update routine
AD EQU    PKRES+$2          ;divided by two adc reading
        ORG    $F02E0
        dc.w  $0000          ;1k peak value, update count value
        dc.w  $0000          ;divided by two adc location

        ORG    $0200

*****  Initialization Routines *****

        INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                                ;set sys clock at 16.78 MHz, disable COP

```



```

***** RAM and Stack Initialization
LDD    #$00FF
STD    RAMBAH           ;store high ram array, bank F
LDD    #$0000
STD    RAMBAL           ;store low ram array, 0000
CLR    RAMMCR           ;enable ram
LDAB   #$0F
TBSK   #$0F             ;set SK to bank F for system stack
LDS    #$02FE           ;put SP in 1k internal SRAM

***** Initialize level 6 autovector address
LDAB   #$00
TBK    #0               ;ek extension pointer = bank0
LDD    #JMPINT          ;load Dacc with interrupt vector addr
STD    $002C            ;store addr to level 6 autovector

***** Initialize the PIT *****
LDAB   #$0F
TBK    #0               ;ek extension pointer = bankf
LDD    #$0616
STD    PICR             ;pirql=6, piv=$16
LDD    #$0101
STD    PISTR            ;set the periodic timer at 62.5msec
ANDP   #$FF1F          ;set interrupt priority to 000

***** QSPI Initialization *****
LDAA   #$08
STAA   QPDR             ;output pcs0/ss* to 0 when asserted
LDAA   #$0F
STAA   QPAR             ;assign QSM port pins to qspi module
LDAA   #$FE
STAA   QDDR             ;assign all QSM pins as outputs except miso

LDD    #$8004           ;mstr, womq=cpol=cpha=0
STD    SPCR0            ;16 bits, 2.10MHz serial baud rate
LDD    #$0300           ;no interrupt generated, no wrap mode
STD    SPCR2            ;newqp=0, endqp=3, queued for 4 trans

***** Fill QSPI Command.ram to write the config registers of the 14489
LDAA   #$C0
STAA   CR0              ;cont=1, bitse=1, pcs0=0, no delays needed
STAA   CR1
STAA   CR2
LDAA   #$40
STAA   CR3              ;cont=0, bitse=1, pcs0=0, no delays needed

***** Fill QSPI Transmit.ram to write the config registers of the 14489
LDAA   #$3F
STD    TR0+1            ;store $3F to tran.ram registers
STD    TR2
STD    TR3+1

***** Turn on the QSPI, this will write to the config registers
***** of the MC14489 drivers
GO     LDD    #$8404
SPIWT STAA   SPCR1      ;turn on spi
LDAA   SPSR             ;after sending data we wait until the
AND    #$80             ;spif bit is set, before we can send more
CMPA   #$80             ;check for spi done
BNE    SPIWT

***** Fill QSPI Command.ram to write the display registers of the 14489
LDAA   #$C0
STAA   CR0              ;cont=1, bitse=1, pcs0=0, no delays needed
STAA   CR1
LDAA   #$40             ;cont=0, bitse=1, pcs0=0, no delays needed
STAA   CR2
STAA   CR4
LDAA   #$80             ;cont=1, bitse=0, pcs0=0, no delays needed
STAA   CR3

```

```

***** Fill QSPI Transmit.ram for display registers of the 14489
***** The beginning LED values will be $00, all of the LEDs will be off
LDD    #$8000
STD    TR0          ;TR0 = $8000
STAA   TR3+1       ;TR1 = $0080
LDD    #$0080      ;TR2 = $0000
STD    TR1          ;TR3 = $XX80
CLRDR  TR4          ;TR4 = $0000
STD    TR2
STD    TR4

LDD    #$0400      ;display registers need 5 transmissions
STD    SPCR2        ;newqp=0, endqp=4

***** ADC Initialization *****
LDD    #$0000
STD    ADCMCR       ;turn on ADC
LDD    #$0003
STD    ADCTL0       ;8-bit, set sample period

***** Initialize the extension registers for the internal ram in bank F
***** Set up the extension registers to point to bank F
LDAB   #$0F         ;load b with $0F
TBEEK  Bacc         ;transfer Bacc to Ek
TBXK   Bacc         ;transfer Bacc to Xk
TBYK   Bacc         ;transfer Bacc to Yk
TBZK   Bacc         ;transfer Bacc to Zk
JMP    RAM          ;jump to internal ram for speed!

***** Start of Internal 1K RAM
RAM    CLR    CNT          ;clear LED update counter
CLR    PK_1K          ;clear 1K peak value

*      Initialization for DSP
ORP    #$0010        ;set saturation mode for Macc
CLRDR  Dacc          ;clear Dacc
TDMSK  Dacc          ;no modulo addressing
LDY    #COEFBS       ;load y with the coef base addr
LDX    #YTRMBS       ;load x with the yterm base addr

LP     CLRDR  Dacc          ; 2 clear Dacc
STD    ADCTL1        ; 6 single 4 conversion, single channel AD0
; writing to the ADCTL1 reg starts conv

*      Divide input x(n) by 2, no overflow problem
LDAA   LJSRR0        ; 6 load Aacc with left jus signed ADC value
ASRA   Dacc          ; 2 divide by 2
STAA   AD            ; 6 store divide by 2 adc value away

*      Check if LEDs need updating
LDAA   CNT           ; 6 load Aacc with count
ADDA   #1            ; 2 add 1 to Aacc
STAA   CNT           ; 6 store new count
BNE    TRAN         ; 6,2 check to see if its time to update
; the LEDs, time = 256 * 668 cycles
; 668 cycles = 40.08usec
; so LED update time is 10.26msec

LDD    #$8404        ; 6 load up d
STD    SPCR1         ; 6 turn on QSPI, send LED data out

TRAN   LDHI   Dacc          ; 8 load h and i multiplier and multiplicand
FLK    CLRDR  Dacc          ; 2 clear Macc
LDE    AD            ; 6 load Eacc with AD

```

```

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_1K      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_1K      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_1K      ; 6 load Dacc with x(n-1)
STED        XN1_1K      ; 8 store x(n) to x(n-1) and
                ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         -4,-4        ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMET                ; 2 transfer Macc to Eacc, truncate
ASLE                ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD1K+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP          ; 2 no operation, due to CPU pipeline
NOP          ; 2 no operation, due to CPU pipeline
LD1K        LDAA        LED_TBL      ; 6 load Aacc with the encoded LED value
                ; from scaled peak LED table

*      Update peak value if needed
CMPA        PK_1K      ; 6 compare value to previous peak value
BLS         DN1K       ; 6,2 branch if not more than peak value
STAA        PK_1K      ; 6 store new peak value
STAA        TR2+1      ; 6 store new value to lk qspi tran.ram

*      Update y(n-1) and y(n-2)
DN1K        LDD         YN1_1K      ; 6 load Dacc with y(n-1)
STED        YN1_1K      ; 8 store Eacc to y(n-1), Dacc to y(n-2)

*****      Loop to generate calculated delay
*****      Clocks = 6 + 8*(N-1)  N >= 1
*****      N is the number put into the B accumulator

WAIT        LDAB        #$3D          ; 61 this loop will create an extra delay
DECB                ; to make a 24.95kHz sampling rate
BNE          WAIT          ; or a 668 cycle sampling period
                ; 486 cycles
NOP                ; 2
NOP                ; 2
NOP                ; 2
JMP         LP            ; 6 jump back to start another conversion

*****      Exceptions/Interrupts      *****
*****      This interrupt is used to decrement the LED bar value
*****      representing the peak value of the lk filter band
INT_RT      PSHM        D,CCR          ;stack Dacc and CCR on stack
LDAA        PK_1K      ;load Aacc with lk peak value
BEQ         DONE       ;equal to 0?, then done

ANDP        #$FEFF      ;clear C bit
RORA                ;rotate right once, decrease peak value
STAA        TR2+1      ;store Aacc to lk Hz qspi tran.ram
STAA        PK_1K      ;store Aacc to lk Hz peak value
LDD         #$8404      ;load up Dacc
STD         SPCR1       ;turn on QSPI, send LED data out

DONE        PULM        D,CCR          ;pull Dacc and CCR from stack
RTI                ;return from interrupt

*****      Location of start of level 6 interrupt, has to be in bank 0
ORG         $A000
JMPINT      JMP         INT_RT

*****      OUTVAL1 is a 256 byte lookup table to convert an
*****      ADC reading to a LED value that can be transmitted to the 14489
*****      Multiplies by two and
*****      Encodes to a scale of +6, +3, 0, -3, -6, -9, -12, -15 dB
*****      INCLUDE 'OUTVAL1.ASM'      ;LED look up table

```

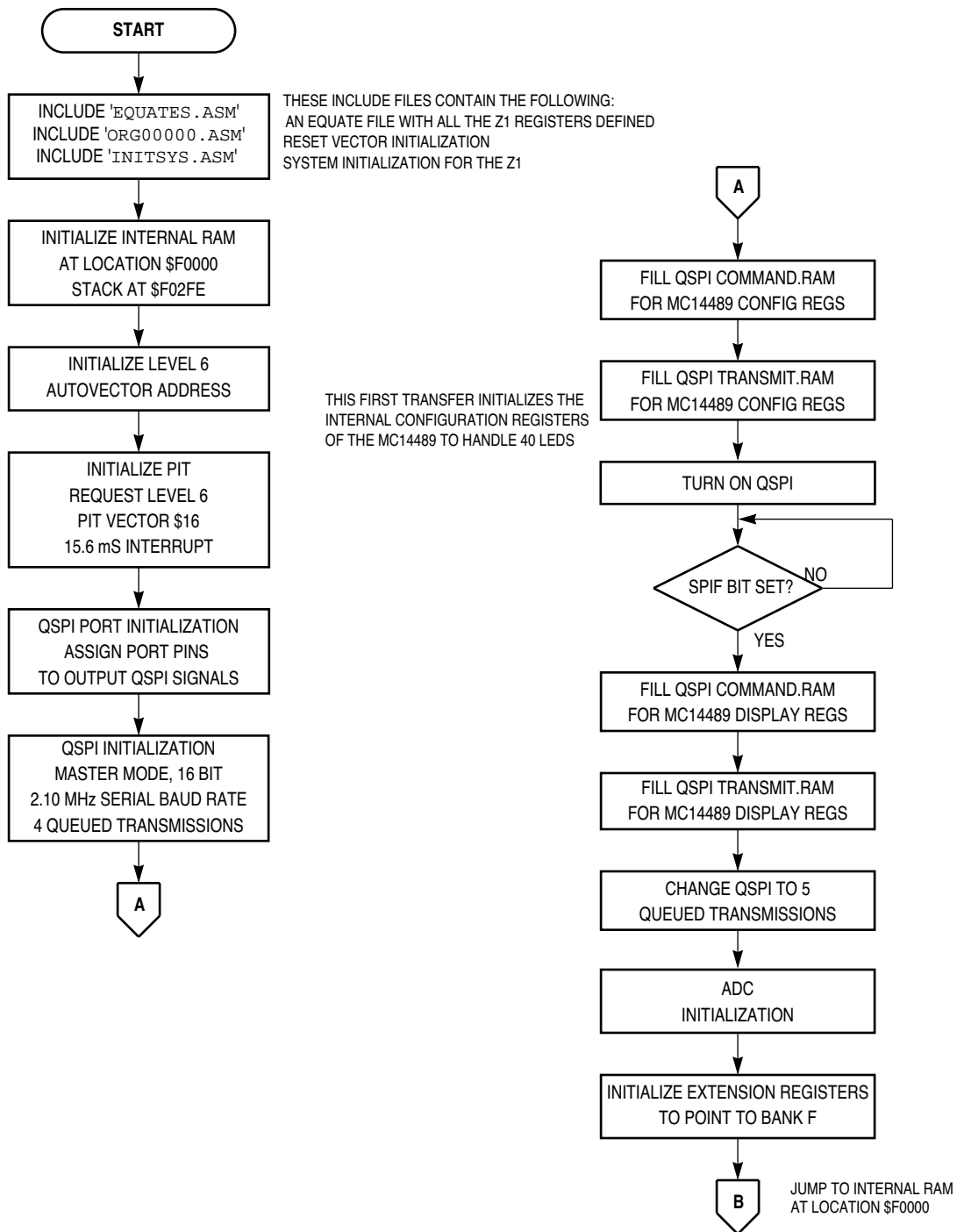
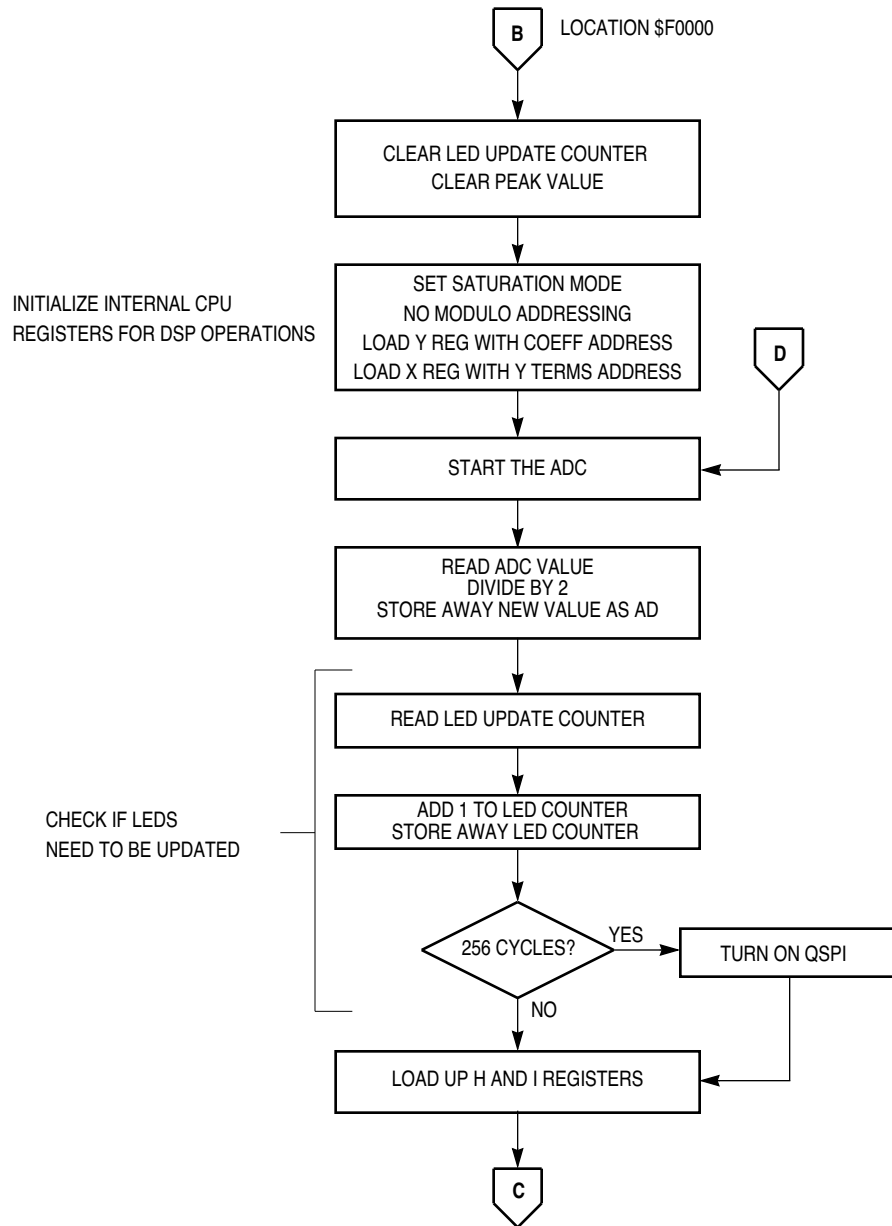


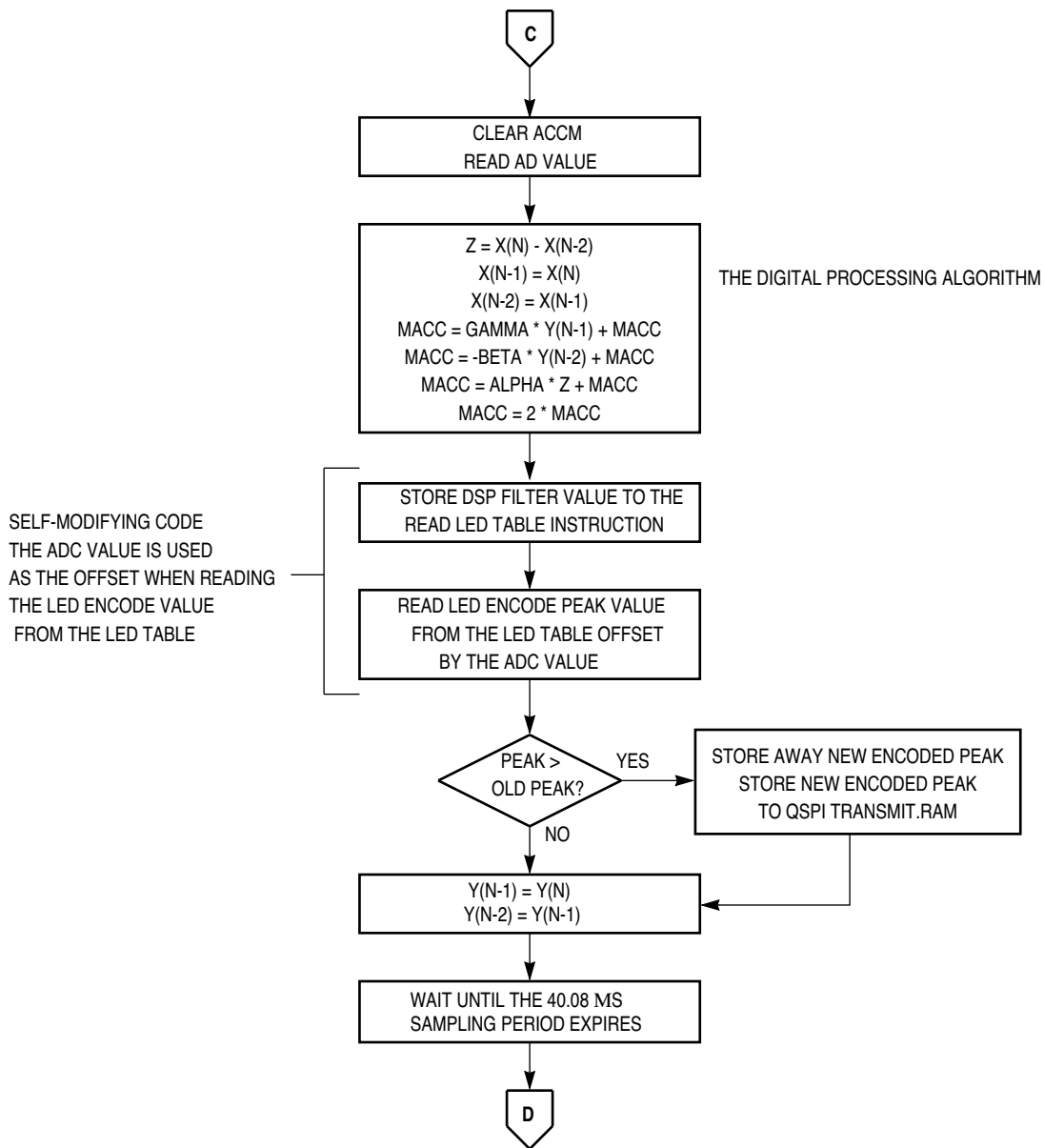
Figure 22 1K_FLTR.ASM Flowchart (Sheet 1 of 4)

AN1233 F20A



AN1233 F22B

Figure 22 1K_FLTR.ASM Flowchart (Sheet 2 of 4)

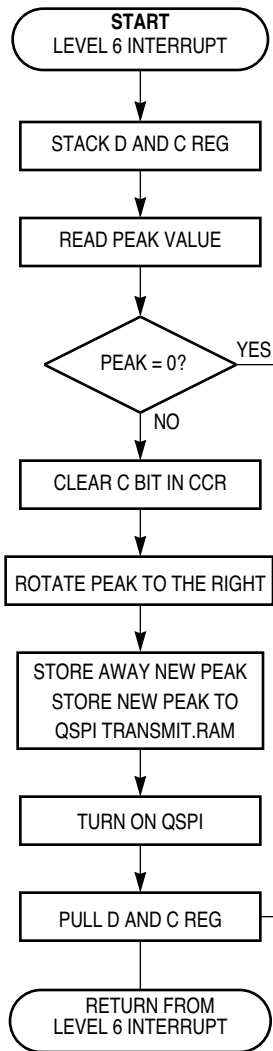


INCLUDE 'OUTVAL1 .ASM'

OUTVAL1.ASM IS A 256 BYTE LOOKUP TABLE TO CONVERT AN ADC VALUE TO AN LED VALUE THAT WILL BE TRANSMITTED TO THE MC14489 THE TABLE WILL MULTIPLY THE VALUE BY TWO AND ENCODE IT TO A SCALE OF +6, +3, 0, -3, -6, -9, -12, AND -15 DB

AN1233 F22C

Figure 22 1K_FLTR.ASM Flowchart (Sheet 3 of 4)



THIS INTERRUPT WILL DECREASE THE LED ENCODED PEAK VALUE. THE DECREASED PEAK VALUE IS THEN TRANSMITTED TO UPDATE THE LED ARRAY.

AN1233 F20C

Figure 22 1K_FLTR.ASM Flowchart (Sheet 4 of 4)

The 5 Band Audio Frequency Analyzer (5BAND_SA.ASM)

The final design of the AFA is simple because of the groundwork that has already been done. **Figure 23** is a flowchart of **5BAND_SA.ASM**. Notice that five iterations of the IIR bandpass filter are executed before control passes to the interrupt routine.

The five bands and their Q values are: 125 Hz – 0.5, 500 Hz – 1.0, 1 kHz – 1.5, 4 kHz – 1.0, and 10 kHz – 0.5. Coefficient values are in the area labeled 'Address of coefficients...' at the beginning of the listing.

The specified Q values were chosen because they produce an appealing frequency display. If sharp filters with high Q values were used, the display would not show the relative differences between the bass, midrange, and treble frequency ranges. Energy associated to one particular frequency is not the primary concern of the AFA design, but rather the energy of an entire frequency band.

Test the code as before with the 1-kHz filter. Sweep a sinusoidal tone across the frequency and watch the appropriate LED array display signal energy. Apply a real time audio signal. Notice the differences between the high and low ends of the audio spectrum, the visible contrast between a bass drum and a cymbal.

5BAND_SA.ASM Code Listing

```
INCLUDE 'EQUATES.ASM' ;table of EQUates for common register addr
INCLUDE 'ORG00000.ASM' ;initialize reset vector

****
Addresses of coefficients for the IIR Filters and initialization
COEFBS EQU $0280 ;base addr of coefficients
GAM_125 EQU COEFBS+$0 ;addr of the gamma coef
BET_125 EQU COEFBS+$2 ;addr of the beta coef
ALP_125 EQU COEFBS+$4 ;addr of the alpha coef
GAM_500 EQU COEFBS+$6 ;addr of the gamma coef
BET_500 EQU COEFBS+$8 ;addr of the beta coef
ALP_500 EQU COEFBS+$A ;addr of the alpha coef
GAM_1K EQU COEFBS+$C ;addr of the gamma coef
BET_1K EQU COEFBS+$E ;addr of the beta coef
ALP_1K EQU COEFBS+$10 ;addr of the alpha coef
GAM_4K EQU COEFBS+$12 ;addr of the gamma coef
BET_4K EQU COEFBS+$14 ;addr of the beta coef
ALP_4K EQU COEFBS+$16 ;addr of the alpha coef
GAM_10K EQU COEFBS+$18 ;addr of the gamma coef
BET_10K EQU COEFBS+$1A ;addr of the beta coef
ALP_10K EQU COEFBS+$1C ;addr of the alpha coef
ORG $F0280
dc.w $7C07 ;125 Hz gamma coef, Q=0.5
dc.w $C3E9 ;125 Hz beta coef, Q=0.5
dc.w $01F4 ;125 Hz alpha coef, Q=0.5
dc.w $7774 ;500 Hz gamma coef, Q=1.0
dc.w $C798 ;500 Hz beta coef, Q=1.0
dc.w $03CB ;500 Hz alpha coef, Q=1.0
dc.w $7257 ;1k Hz gamma coef, Q=1.5
dc.w $C9F0 ;1k Hz beta coef, Q=1.5
dc.w $04F7 ;1k Hz alpha coef, Q=1.5
dc.w $2C13 ;4k Hz gamma coef, Q=1.0
dc.w $ED7A ;4k Hz beta coef, Q=1.0
dc.w $16BC ;4k Hz alpha coef, Q=1.0
dc.w $CA66 ;10k Hz gamma coef, Q=0.5
dc.w $FD FE ;10k Hz beta coef, Q=0.5
dc.w $1EFE ;10k Hz alpha coef, Q=0.5
```



```

***** Addresses of filter terms for the x(n) terms and initialization
XTRMBS EQU $02A0 ;base addr of x(n) filter terms
XN1_125 EQU XTRMBS+$0 ;x(n-1)
XN2_125 EQU XTRMBS+$2 ;x(n-2)
XN1_500 EQU XTRMBS+$4 ;x(n-1)
XN2_500 EQU XTRMBS+$6 ;x(n-2)
XN1_1K EQU XTRMBS+$8 ;x(n-1)
XN2_1K EQU XTRMBS+$A ;x(n-2)
XN1_4K EQU XTRMBS+$C ;x(n-1)
XN2_4K EQU XTRMBS+$E ;x(n-2)
XN1_10K EQU XTRMBS+$10 ;x(n-1)
XN2_10K EQU XTRMBS+$12 ;x(n-2)
ORG $F02A0
dc.w $0000 ;125 Hz x(n-1)
dc.w $0000 ;125 Hz x(n-2)
dc.w $0000 ;500 Hz x(n-1)
dc.w $0000 ;500 Hz x(n-2)
dc.w $0000 ;1k Hz x(n-1)
dc.w $0000 ;1k Hz x(n-2)
dc.w $0000 ;1k Hz x(n-1)
dc.w $0000 ;1k Hz x(n-2)
dc.w $0000 ;1k Hz x(n-1)
dc.w $0000 ;1k Hz x(n-2)

```

```

***** Addresses of filter terms for the y(n) terms and initialization
YTRMBS EQU $02C0 ;base addr of y(n) filter terms
YN1_125 EQU YTRMBS+$0 ;y(n-1)
YN2_125 EQU YTRMBS+$2 ;y(n-2)
X_2_125 EQU YTRMBS+$4 ;x(n) - x(n-2), stored here for mac
YN1_500 EQU YTRMBS+$6 ;y(n-1)
YN2_500 EQU YTRMBS+$8 ;y(n-2)
X_2_500 EQU YTRMBS+$A ;x(n) - x(n-2), stored here for mac
YN1_1K EQU YTRMBS+$C ;y(n-1)
YN2_1K EQU YTRMBS+$E ;y(n-2)
X_2_1K EQU YTRMBS+$10 ;x(n) - x(n-2), stored here for mac
YN1_4K EQU YTRMBS+$12 ;y(n-1)
YN2_4K EQU YTRMBS+$14 ;y(n-2)
X_2_4K EQU YTRMBS+$16 ;x(n) - x(n-2), stored here for mac
YN1_10K EQU YTRMBS+$18 ;y(n-1)
YN2_10K EQU YTRMBS+$1A ;y(n-2)
X_2_10K EQU YTRMBS+$1C ;x(n) - x(n-2), stored here for mac
ORG $F02C0
dc.w $0000 ;125 Hz y(n-1)
dc.w $0000 ;125 Hz y(n-2)
dc.w $0000 ;125 Hz [ x(n) - x(n-2) ]
dc.w $0000 ;500 Hz y(n-1)
dc.w $0000 ;500 Hz y(n-2)
dc.w $0000 ;500 Hz [ x(n) - x(n-2) ]
dc.w $0000 ;1k Hz y(n-1)
dc.w $0000 ;1k Hz y(n-2)
dc.w $0000 ;1k Hz [ x(n) - x(n-2) ]
dc.w $0000 ;4k Hz y(n-1)
dc.w $0000 ;4k Hz y(n-2)
dc.w $0000 ;4k Hz [ x(n) - x(n-2) ]
dc.w $0000 ;10k Hz y(n-1)
dc.w $0000 ;10k Hz y(n-2)
dc.w $0000 ;10k Hz [ x(n) - x(n-2) ]

```

```

***** Addresses of various temporary variables and initialization
PKRES EQU $02E0 ;base addr of filter result storage
PK_125 EQU PKRES+$0 ;peak value for 125 Hz
PK_500 EQU PKRES+$1 ;peak value for 500 Hz
PK_1K EQU PKRES+$2 ;peak value for 1k Hz
PK_4K EQU PKRES+$3 ;peak value for 4k Hz
PK_10K EQU PKRES+$4 ;peak value for 10k Hz
CNT EQU PKRES+$6 ;count value for LED qsapi update routine
AD EQU PKRES+$8 ;divided by two adc reading
ORG $F02E0
dc.w $0000 ;125 peak value, 500 peak value
dc.w $0000 ;1k peak value, 4k peak value
dc.w $0000 ;10k peak value
dc.w $0000 ;update count value
dc.w $0000 ;divided by two adc reading

ORG $0200

```

```

***** Initialization Routines *****

INCLUDE 'INITSYS.ASM' ;initially set EK=F, XK=0, YK=0, ZK=0
                       ;set sys clock at 16.78 MHz, disable COP

***** RAM and Stack Initialization *
LDD    #$00FF
STD    RAMBAH          ;store high ram array, bank F
LDD    #$0000
STD    RAMBAL          ;store low ram array, 0000
CLR    RAMMCR          ;enable ram
LDAB   #$0F
TBSK   ;set SK to bank F for system stack
LDS    #$02FE          ;put SP in 1k internal SRAM

***** Initialize level 6 autovector address
LDAB   #$00
TBK    ;ek extension pointer = bank0
LDD    #JMPINT         ;load Dacc with interrupt vector addr
STD    $002C           ;store addr to level 6 autovector

***** Initialize the PIT *****
LDAB   #$0F
TBK    ;ek extension pointer = bankf
LDD    #$0616
STD    PICR            ;pirql=6, piv=$16
LDD    #$0101
STD    PITR            ;set the periodic timer at 62.5msec
ANDP   #$FF1F         ;set interrupt priority to 000

***** QSPI Initialization *****
LDAA   #$08
STAA   QPDR            ;output pcs0/ss* to 0 when asserted
LDAA   #$0F
STAA   QPAR            ;assign QSM port pins to qspi module
LDAA   #$FE
STAA   QDDR            ;assign all QSM pins as outputs except miso

LDD    #$8004          ;mstr, womq=cpol=cpha=0
STD    SPCR0           ;16 bits, 2.10MHz serial baud rate
LDD    #$0300          ;no interrupt generated, no wrap mode
STD    SPCR2           ;newqp=0, endqp=3, queued for 4 trans

***** Fill QSPI Command.ram to write the config registers of the 14489
LDAA   #$C0
STAA   CR0             ;cont=1, bitse=1, pcs0=0, no delays needed
STAA   CR1
STAA   CR2
LDAA   #$40
STAA   CR3             ;cont=0, bitse=1, pcs0=0, no delays needed

***** Fill QSPI Transmit.ram to write the config registers of the 14489
LDAA   #$3F
STD    TR0+1           ;store $3F to tran.ram registers
STD    TR2
STD    TR3+1

***** Turn on the QSPI, this will write to the config registers
***** of the MCL14489 drivers
GO     LDD    #$8404
SPIWT  STAA   SPCR1     ;turn on spi
LDAA   SPSR            ;after sending data we wait until the
ANDA   #$80            ;spif bit is set, before we can send more
CMPA   #$80            ;check for spi done
BNE    SPIWT

***** Fill QSPI Command.ram to write the display registers of the 14489
LDAA   #$C0
STAA   CR0             ;cont=1, bitse=1, pcs0=0, no delays needed
STAA   CR1
LDAA   #$40            ;cont=0, bitse=1, pcs0=0, no delays needed
STAA   CR2
STAA   CR4
LDAA   #$80            ;cont=1, bitse=0, pcs0=0, no delays needed
STAA   CR3

```

```

***** Fill QSPI Transmit.ram for display registers of the 14489
***** The beginning LED values will be $00, all of the LEDs will be off
LDD    #$8000
STD    TR0          ;TR0 = $8000
STAA   TR3+1       ;TR1 = $0080
LDD    #$0080      ;TR2 = $0000
STD    TR1          ;TR3 = $XX80
CLRDL  CLRD        ;TR4 = $0000
STD    TR2
STD    TR4

LDD    #$0400      ;display registers need 5 transmissions
STD    SPCR2       ;newqp=0, endqp=4

***** ADC Initialization *****
LDD    #$0000
STD    ADCMCR      ;turn on ADC
LDD    #$0003
STD    ADCTL0      ;8-bit, set sample period

***** Initialize the extension registers for the internal ram in bank F
***** Set up the extension registers to point to bank F
LDAB   #$0F        ;load b with $0F
TBEEK  ;transfer Bacc to Ek
TBXK   ;transfer Bacc to Xk
TBYK   ;transfer Bacc to Yk
TBZK   ;transfer Bacc to Zk
JMP    RAM         ;jump to internal ram for speed!

***** Start of Internal 1K RAM
RAM    ORG    $F0000
CLR    CNT         ;clear LED update counter
CLR    PK_125      ;clear 125 peak value
CLR    PK_500      ;clear 500 peak value
CLR    PK_1K       ;clear 1k peak value
CLR    PK_4K       ;clear 4k peak value
CLR    PK_10K      ;clear 10k peak value
CLRW   AD          ;clear AD

*      Initialization for DSP
ORP    #$0010      ;set saturation mode for Macc
CLRDL  CLRD        ;clear Dacc
TDMSK  ;no modulo addressing

LP     LDY    #COEFBS ; 4 load y with the coef base addr
LDX    #YTRMBS     ; 4 load x with the yterm base addr
LDHI   ; 8 load h and i multiplier and multiplicand
CLRDL  ; 2 clear Dacc
STD    ADCTL1      ; 6 single 4 conversion, single channel AD0
;      writing to the ADCTL1 reg starts conv

*      Divide input x(n) by 2, no overflow problem
LDAA   LJSRR0      ; 6 load Aacc with left jus signed ADC value
ASRA   ; 2 divide by 2
STAA   AD          ; 6 store divide by 2 adc value away

*      Check if LEDs need updating
LDAA   CNT         ; 6 load Aacc with count
ADDA   #1          ; 2 add 1 to Aacc
STAA   CNT         ; 6 store new count
BNE    F125        ; 6,2 check to see if its time to update
;      the LEDs, time = 256 * 668 cycles
;      668 cycles = 40.08usec
;      so LED update time is 10.26msec
LDD    #$8404      ; 6 load up Dacc
STD    SPCR1       ; 6 turn on QSPI, send LED data out

***** Start of the 125 Hz routine
F125  CLRML  ; 2 clear Macc
LDE   AD          ; 6 load Eacc with AD

```

```

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_125      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_125      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_125      ; 6 load Dacc with x(n-1)
STED        XN1_125      ; 8 store x(n) to x(n-1) and
                ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         2,2          ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMER        ; 6 transfer Macc to Eacc, round for converg
ASLE        ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD125+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP         ; 2 no operation, due to CPU pipeline
NOP         ; 2 no operation, due to CPU pipeline
LD125 LDAA    LED_TBL     ; 6 load Aacc with the encoded LED value
                ; from scaled peak LED table

*      Update peak value if needed
CMPA        PK_125       ; 6 compare value to previous peak value
BLS         DN125        ; 6,2 branch if not more than peak value
STAA        PK_125       ; 6 store new peak value
STAA        TR4+1        ; 6 store new value to 125 qspi tran.ram

*      Update y(n-1) and y(n-2)
DN125 LDD     YN1_125     ; 6 load Dacc with y(n-1)
STED        YN1_125     ; 8 store Eacc to y(n-1), Dacc to y(n-2)

***** Start of the 500 Hz DSP routine
F500 CLRM      ; 2 clear Macc
LDE        AD          ; 6 load Eacc with AD

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_500      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_500      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_500      ; 6 load Dacc with x(n-1)
STED        XN1_500      ; 8 store x(n) to x(n-1) and
                ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         2,2          ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMET        ; 2 transfer Macc to Eacc, truncate
ASLE        ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD500+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP         ; 2 no operation, due to CPU pipeline
NOP         ; 2 no operation, due to CPU pipeline
LD500 LDAA    LED_TBL     ; 6 load Aacc with the encoded LED value
                ; from scaled peak LED table

*      Update peak value if needed
CMPA        PK_500       ; 6 compare value to previous peak value
BLS         DN500        ; 6,2 branch if not more than peak value
STAA        PK_500       ; 6 store new peak value
STAA        TR4          ; 6 store new value to 500 qspi tran.ram

*      Update y(n-1) and y(n-2)
DN500 LDD     YN1_500     ; 6 load Dacc with y(n-1)
STED        YN1_500     ; 8 store Eacc to y(n-1), Dacc to y(n-2)

***** Start of the 1k Hz routine
F1K CLRM      ; 2 clear Macc
LDE        AD          ; 6 load Eacc with AD

```

```

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_1K      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_1K      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_1K      ; 6 load Dacc with x(n-1)
STED        XN1_1K      ; 8 store x(n) to x(n-1) and
                ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         2,2          ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMET        ; 2 transfer Macc to Eacc, truncate
ASLE        ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD1K+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP         ; 2 no operation, due to CPU pipeline
NOP         ; 2 no operation, due to CPU pipeline
LD1K        LDAA        LED_TBL      ; 6 load Aacc with the encoded LED value
                ; from scaled peak LED table

*      Update peak value if needed
CMPA        PK_1K       ; 6 compare value to previous peak value
BLS         DN1K        ; 6,2 branch if not more than peak value
STAA        PK_1K       ; 6 store new peak value
STAA        TR2+1       ; 6 store new value to lk qspi tran.ram

*      Update y(n-1) and y(n-2)
DN1K        LDD         YN1_1K      ; 6 load Dacc with y(n-1)
STED        YN1_1K      ; 8 store Eacc to y(n-1), Dacc to y(n-2)

*****      Start of the 4k Hz routine
F4K         CLRM        ; 2 clear Macc
LDE         AD          ; 6 load Eacc with AD

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_4K      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_4K      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_4K      ; 6 load Dacc with x(n-1)
STED        XN1_4K      ; 8 store x(n) to x(n-1) and
                ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         2,2          ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMET        ; 2 transfer Macc to Eacc, truncate
ASLE        ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD4K+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP         ; 2 no operation, due to CPU pipeline
NOP         ; 2 no operation, due to CPU pipeline
LD4K        LDAA        LED_TBL      ; 6 load Aacc with the encoded LED value
                ; from scaled peak LED table

*      Update peak value if needed
CMPA        PK_4K       ; 6 compare value to previous peak value
BLS         DN4K        ; 6,2 branch if not more than peak value
STAA        PK_4K       ; 6 store new peak value
STAA        TR2         ; 6 store new value to 4k qspi tran.ram

*      Update y(n-1) and y(n-2)
DN4K        LDD         YN1_4K      ; 6 load Dacc with y(n-1)
STED        YN1_4K      ; 8 store Eacc to y(n-1), Dacc to y(n-2)

*****      Start of the 10k Hz routine
F10K        CLRM        ; 2 clear Macc
LDE         AD          ; 6 load Eacc with AD

```

```

*      Digital processing algorithm
TED          ; 2 transfer Eacc to Dacc
SUBD        XN2_10K      ; 6 Dacc = x(n) - x(n-2)
STD         X_2_10K      ; 6 store Dacc to [x(n) - x(n-2)] addr
LDD         XN1_10K      ; 6 load Dacc with x(n-1)
STED        XN1_10K      ; 8 store x(n) to x(n-1) and
                    ; store x(n-1) to x(n-2)

MAC         2,2          ;12 gamma*(yn1)+Macc=Macc
MAC         2,2          ;12 beta*(yn2)+Macc=Macc
MAC         2,2          ;12 alpha*[x(n)-x(n-2)]+Macc=Macc
TMET        ; 2 transfer Macc to Eacc, truncate
ASLE        ; 2 multiply Eacc by 2

*      Get LED encode value from look-up table
TED          ; 2 transfer Eacc to Dacc
STAA        LD10K+3      ; 6 Dacc high byte -> instruction ldaa $03??
NOP         ; 2 no operation, due to CPU pipeline
NOP         ; 2 no operation, due to CPU pipeline
LD10K      LDAA        LED_TBL      ; 6 load Aacc with the encoded LED value
                    ; from scaled peak LED table

*      Update peak value
CMPA        PK_10K      ; 6 compare value to previous peak value
BLS         DN10K      ; 6,2 branch if not more than peak value
STAA        PK_10K      ; 6 store new peak value
STAA        TR1         ; 6 store new value to 10k qspi tran.ram

*      Update y(n-1) and y(n-2)
DN10K      LDD         YN1_10K      ; 6 load Dacc with y(n-1)
STED        YN1_10K      ; 8 store Eacc to y(n-1), Dacc to y(n-2)
NOP

END        JMP         LP          ; 6 jump back to start another conversion

```

```

**** Exceptions/Interrupts ****
**** This interrupt is used to decrement each LED bar value
**** representing the peak value of each filter band
INT_RT PSHM      D,CCR          ;stack Dacc and CCR on stack

CK125  LDAA      PK_125        ;load Aacc with 125 peak value
      BEQ      CK500          ;equal to 0?, then CK500
      ANDP     #$FEFF         ;clear C bit
      RORA     ;rotate right once, decrease peak value
      STAA     TR4+1          ;store Aacc to 125 Hz qsapi tran.ram
      STAA     PK_125         ;store Aacc to 125 Hz peak value

CK500  LDAA      PK_500        ;load Aacc with 500 peak value
      BEQ      CK1K           ;equal to 0?, then CK1K
      ANDP     #$FEFF         ;clear C bit
      RORA     ;rotate right once, decrease peak value
      STAA     TR4            ;store Aacc to 500 Hz qsapi tran.ram
      STAA     PK_500         ;store Aacc to 500 Hz peak value

CK1K   LDAA      PK_1K         ;load Aacc with 1k peak value
      BEQ      CK4K           ;equal to 0?, then CK4K
      ANDP     #$FEFF         ;clear C bit
      RORA     ;rotate right once, decrease peak value
      STAA     TR2+1          ;store Aacc to 1k Hz qsapi tran.ram
      STAA     PK_1K          ;store Aacc to 1k Hz peak value

CK4K   LDAA      PK_4K         ;load Aacc with 4k peak value
      BEQ      CK10K          ;equal to 0?, then CK10K
      ANDP     #$FEFF         ;clear C bit
      RORA     ;rotate right once, decrease peak value
      STAA     TR2            ;store Aacc to 4k Hz qsapi tran.ram
      STAA     PK_4K          ;store Aacc to 4k Hz peak value

CK10K  LDAA      PK_10K        ;load Aacc with 10k peak value
      BEQ      UPDATE         ;equal to 0?, then UPDATE
      ANDP     #$FEFF         ;clear C bit
      RORA     ;rotate right once, decrease peak value
      STAA     TR1            ;store Aacc to 10k Hz qsapi tran.ram
      STAA     PK_10K         ;store Aacc to 10k Hz peak value

UPDATE LDD      #$8404         ;load up Dacc
      STD      SPCR1          ;turn on QSAPI, send LED data out

DONE   PULM     D,CCR          ;pull Dacc and CCR from stack
      RTI                      ;return from interrupt

**** Location of start of level 6 interrupt, has to be in bank 0
      ORG      $A000
JMPINT JMP      INT_RT

**** OUTVAL1 is a 256 byte lookup table to convert an
**** ADC reading to a LED value that can be transmitted to the 14489
**** Multiplies by two and
**** Encodes to a scale of +6, +3, 0, -3, -6, -9, -12, -15 dB
      INCLUDE 'OUTVAL1.ASM'    ;LED Look up table

```

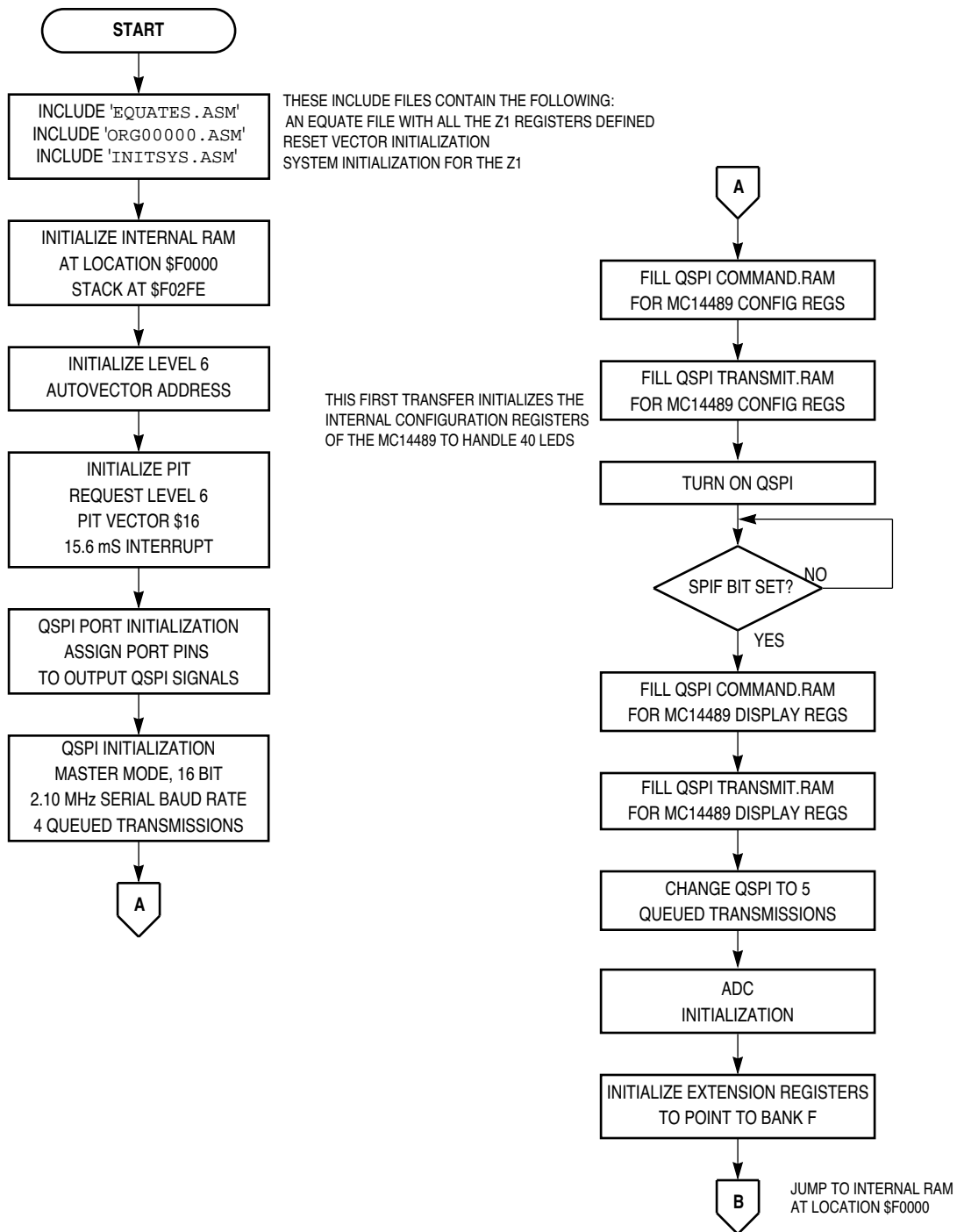
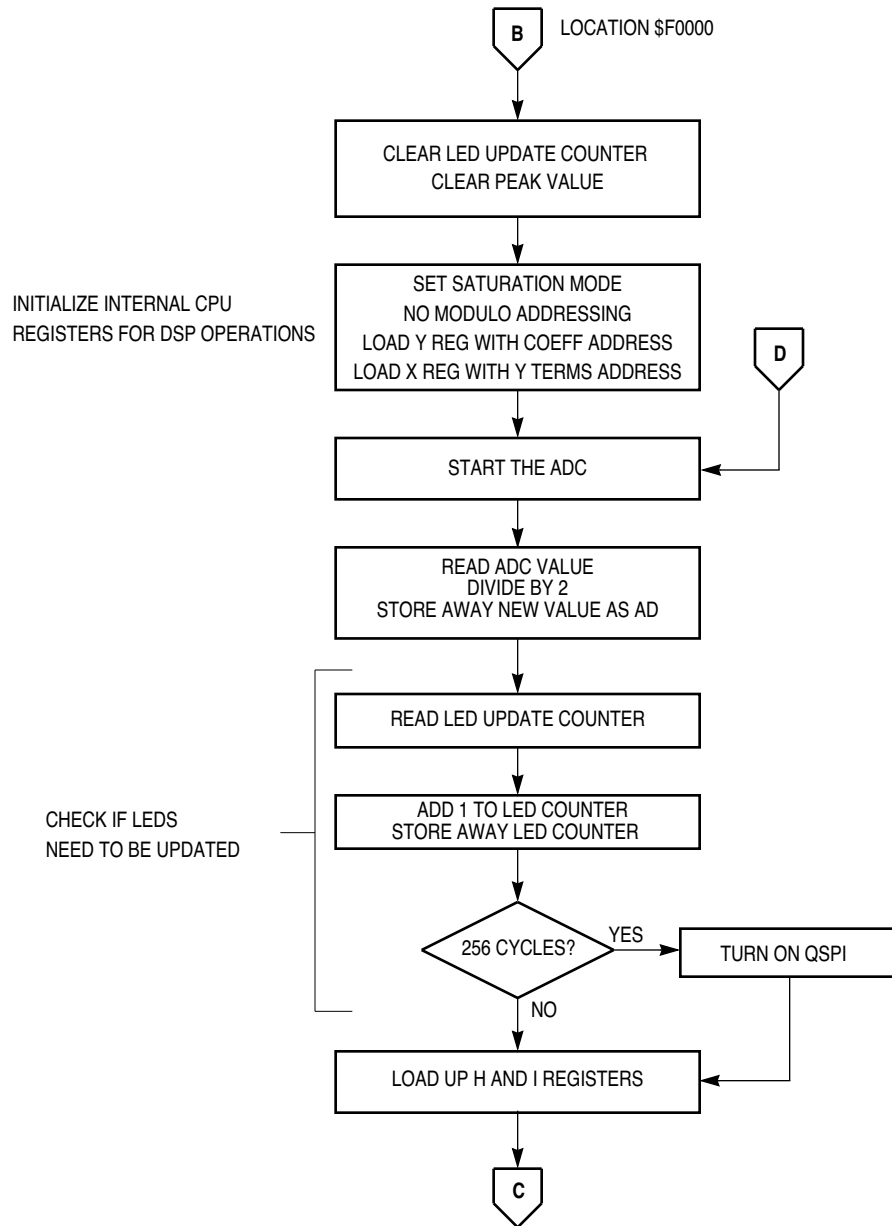


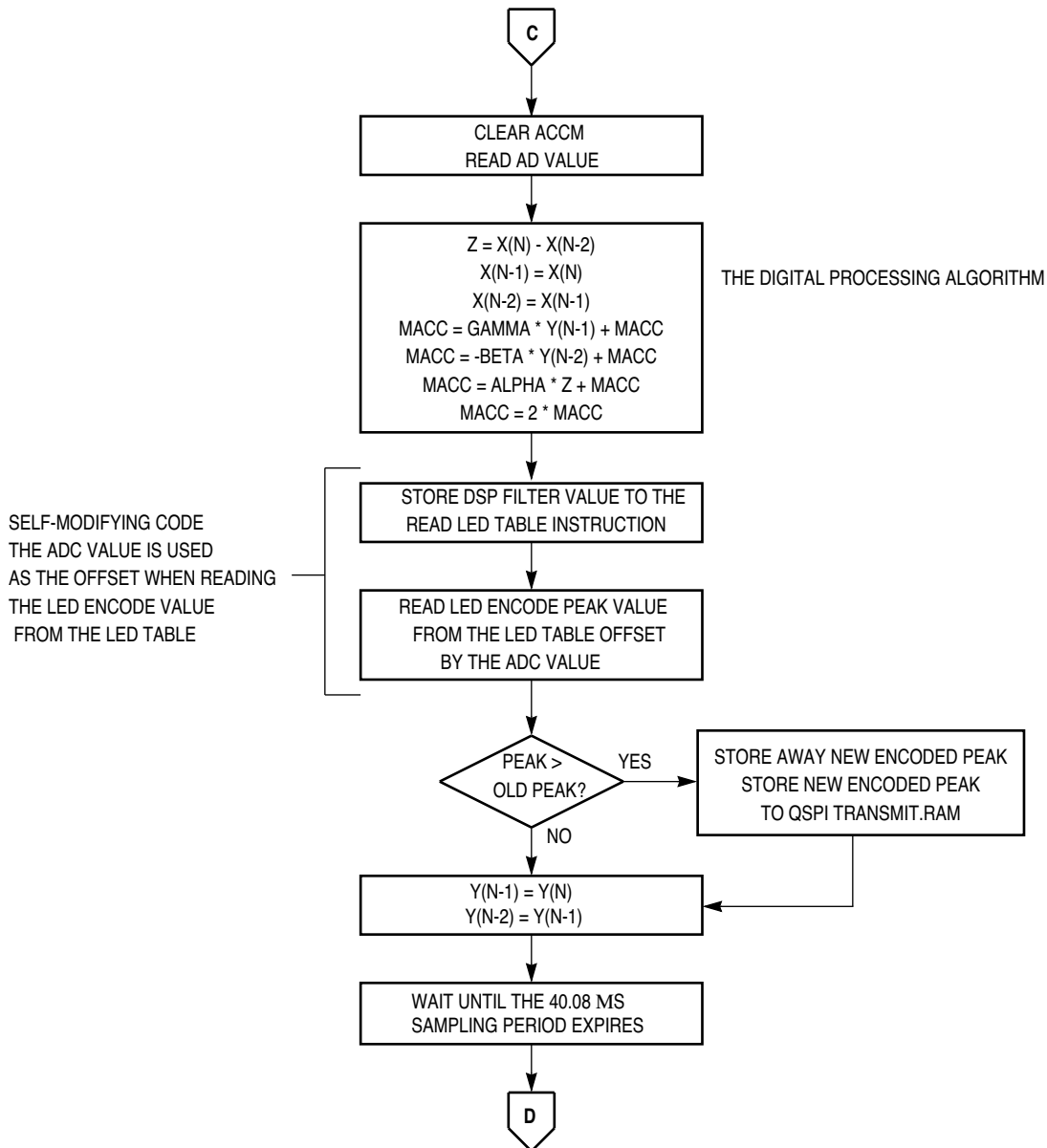
Figure 23 5BAND_SA.ASM Flowchart (Sheet 1 of 4)

AN1233 F20A



AN1233 F22B

Figure 23 5BAND_SA.ASM Flowchart (Sheet 2 of 4)

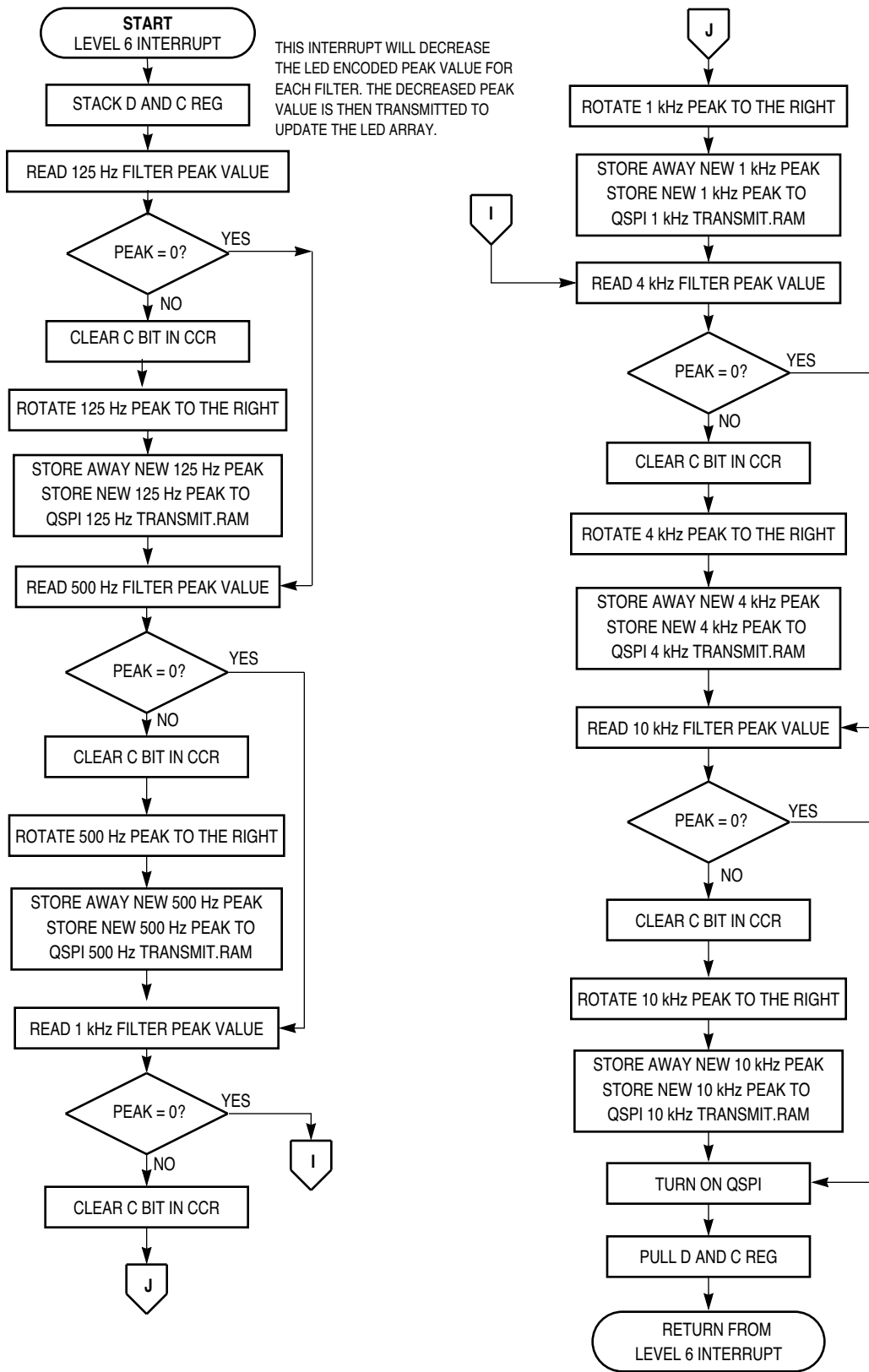


INCLUDE 'OUTVAL1 . ASM'

OUTVAL1.ASM IS A 256 BYTE LOOKUP TABLE TO CONVERT AN ADC VALUE TO AN LED VALUE THAT WILL BE TRANSMITTED TO THE MC14489 THE TABLE WILL MULTIPLY THE VALUE BY TWO AND ENCODE IT TO A SCALE OF +6, +3, 0, -3, -6, -9, -12, AND -15 DB

AN1233 F22C

Figure 23 5BAND_SA.ASM Flowchart (Sheet 3 of 4)



AN1233 F23

Figure 23 5BAND_SA.ASM Flowchart (Sheet 4 of 4)

CONCLUSION


This application note is intended to give designers some insight concerning the use of digital signal processing algorithms with a microcontroller. The finished project is flexible enough to permit experimenting with different filters and LED output displays. DSP allows the experimenter to make on-the-fly changes in filter response by changing the coefficients.

REFERENCES

The following Motorola documents are referred to in this application note.

- *M68HC16Z1EVB User's Manual* (M68HC16Z1EVB/D)
- *MC68HC16Z1 User's Manual* (MC68HC16Z1UM/D)
- *CPU16 Reference Manual* (CPU16RM/AD)
- *QSM Reference Manual* (QSMRM/AD)
- *ADC Reference Manual* (ADCRM/AD)
- MC14489 Data Sheet (MC14489/D)

These items can be obtained through a Motorola Sales Office or Literature Distribution Center.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.  **MOTOROLA** is a registered trademark of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

TO OBTAIN ADDITIONAL PRODUCT INFORMATION:

USA/EUROPE: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036. 1-800-441-2447

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, Toshikatsu Otsuki,
6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 03-3521-8315

HONG KONG: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

MFAX: RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609

INTERNET: <http://www.mot.com>



MOTOROLA