soundart

APPLICATION NOTE
AN002

monosynth: a monophonic synthesizer for **chameleon**

revision 0 | 05.2002
chameleon S.D.K. v1.2

Introduction

MonoSynth is a complete monophonic synthesizer application for Soundart's Chameleon with two wavetable oscillators, a white noise generator, a mixer, a resonant lowpass filter with cutoff envelope (ADSR), an amplifier with gain envelope (ADSR) and two (stereo) delay effect units.

The source code of the application is included as an example in the Chameleon Software Development Kit (SDK) from version 1.2

This example covers practically all aspects necessary to implement a typical synthesizer. These are some of the key characteristics:

- Use of a modular-like DSP processing framework.
- Included DSP code implementing generic blocks of frequently used modules: oscillators, filter, vca, envelopes, delay lines...
- Fast ColdFire to DSP, interrupt based, generic communication framework.
- DSP simulator/profiling support code (using conditional assembly directives).
- Realtime computation of DSP parameters in the ColdFire using standard floating point functions in C.
- Realtime processing of incoming MIDI data, supporting NoteOn, NoteOff, PitchBend, Controllers, ProgramChange and SysEx messages.
- Realtime generation of MIDI controller data using the panel knobs.
- Programmable MIDI-thru-merge engine (MIDI input messages routed to MIDI output in realtime together with the internal generated ones).
- MIDIFile sequencer used to play the included demo song.
- Easily extendable menu based panel operation framework.
- Possibility of assigment of the panel knobs to any defined parameter.
- Use of user flash memory to store until 128 sound presets and non volatil configuration data (MIDI device ID, receive MIDI channel, ...)

- Implementation of typical preset dump and request MIDI system exclusive messages.

- Use of object oriented C++ classes in the ColdFire side to make easy the addition of new features, and to promote the reuse of commonly used code.

- Multitasking, object oriented, message-queue and priority based implementation of the different conceptual elements involved.

- Fast access to catalogued parameters by ID using a database-like parameter container (using a fast hash table).

- Easy to change mapping tables to assign MIDI controllers to internal parameters.

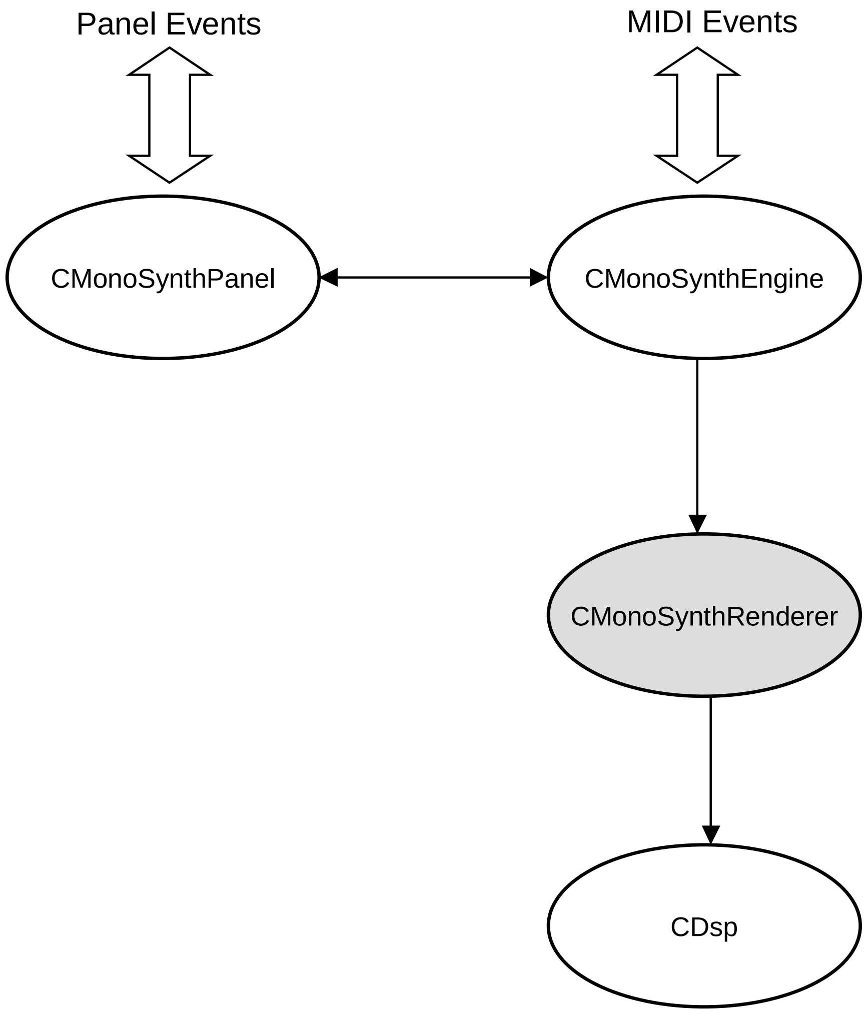
ColdFire Side

The ColdFire part of the example is composed by a number of C++ files which define the objects used in the application. The code uses class inheritance to define commonly used routines in base classes.

The application is divided conceptually in three different blocks: the **panel handler** (*CMonoSynthPanel*), the **MIDI handler** (*CMonoSynthEngine*) and the **DSP handler** (*CDsp*). Each one of these handlers are really realtime priorized RTEMS tasks with RTEMS message queues to allow easily safe inter-task communications. The base class of these blocks (*CMessageHandler*) encapsulates this functionality offering generic functions members like *SendMessage* or the virtual *OnMessage* to allow derived classes to implement easily new message types (derived from class *CMessage*).

Giving a general outline, the class *CMonoSynthRenderer* contains all available application parameters as private data members. During the initialization, these members are made globally accessible by assigning a unique identifier (ID) to each one, and using a static *CHashTable* object. The class *CMonoSynthPanel* allows the user to change the parameter values and when this happens, it notifies to the *CMonoSynthEngine* class to update the parameter value in the DSP (using *CMonoSynthRenderer*). *CMonoSynthEngine* can also change parameter values by loading presets from flash memory, or by receiving MIDI data. The object *CMonoSynthRenderer* is contained inside the *CMonoSynthEngine*; it has a private *CDsp* object inside to allow direct communication with the DSP. Thus, the *CMonSynthEngine* is the only class which has access to the DSP.

Figure 1
Main ColdFire C++ classes interaction



Panel Handler

The **panel handler** is used to manage the user interaction with Chameleon panel elements. *CMonoSynthPanel* is derived from *CPanel*, which in turn is derived from *CMessageHandler*. Therefore *CMonoSynthPanel* is implemented as a RTEMS task runnning in parallel with the other tasks. *CPanel* contains a reference to the panel driver, receives input events from the local message queue using the standard way proposed (*OnMessage*), and includes function members to modify output panel elements (LCD and LEDs).

The **panel handler** sends and receives messages to and from the **MIDI handler** to inform and to be informed mutually of the changes made, using defined messages like ParameterChanged or PresetChanged.

To implement different application operating modes and menu structures, it has been defined the base class *CPanelModeHandler*. Several derived objects from this class are used by the **panel handler** (only one of them at a time) to easily change the operation mode of the application without keep a lot of global variables. This class defines a number of common virtual function members to allow derived classes to change the default behaviour used to implement the different operating modes. The MonoSynth application uses these different operating modes/states:

- Init mode (*CMonoSynthPanelModeInit*): this mode is the first used when the application boots. It shows a welcome message and waits a bit during the initialization of the rest of components. After initialization is completed, the application change to the preset mode.

```
   mono synth
      v1.0
```

- Preset mode (*CMonoSynthPanelModePreset*): this is the typical operating mode used to navigate across the 128 presets stored in non volatile memory. The display shows the current preset number and name, and a flashing asterisk if any of the preset parameters has been modified to notice that. The LEDs located above the panel knobs will be turned on if the preset has assigned that knob to any parameter. If the user turn left or right one the assigned knobs, the application changes to the assign mode.

```
Preset 001 *
Fat Bassey
```

  The user can change the current preset using the panel encoder or the value up and down keys. The edit key puts the panel in the edit mode, and the group up or group down keys changes to the midi dump mode.

  Pressing together the shift and part down keys, the Preset mode changes to the demo playing state which loads the midifile DEMO.MID supplied with the source code and included in the application using the new bin2header utility. Once the midifile is loaded using the MidiShare Player library function MidiFileLoadFromMemory, it is played in loop mode (using the SetLoopPlayer and StartPlayer functions) until the user press the shift key again, exiting then from the demo playing state.

- Edit mode (*CMonoSynthPanelModeEdit*): This mode defines a menu structure to allow the navigation and modification of all available preset parameters using the keys and the encoder. The display is divided in four areas: The upper left area shows the current parameter group name; the upper right area shows the current parameter page name (inside the current group); the lower left area shows the current parameter name (inside the current page in the current group); and the lower right area shows the current parameter value.

```
OSCILLATOR: Osc1
Waveform  [ Saw]
```

The user can move the encoder or press the value up and down keys to change the current parameter value. The param up and down keys change the current parameter inside the current page selected. The page up and down keys are used to change the current parameter page inside the current parameter group selected. And finally, the group up and down keys are used to change the current parameter group.

The menu structure is defined during the initialization of *CMonoSynthPanelModeEdit* using *CMenuItem* objects. Using this base class for menu items, each group item keeps the current page selected, and each page item keeps the current parameter item selected.

The MonoSynth application defines the following menu structure:

GROUP	PAGE	PARAM
OSCILLATOR	Osc1	Waveform
		Transpose
	Osc2	Waveform
		Transpose
		Detune
MIXER		Osc1Vol
		Osc2Vol
		NoiseVol
		InputVol
FILTER		Cutoff
		Resonance
		EnvDepth
		KeyDepth
ENV	Filter	Attack
		Decay
		Sustain
		Release
	Amp	Attack
		Decay
		Sustain
		Release
AMP		EnvDepth
		KeyDepth
EFX	Delay Left	DelayLev
		DelayTime
		DelayFb
	Delay Right	DelayLev
		DelayLev
		DelayFb

GLOBAL	InputThru
	MidiChannel
	MidiThru
	Midi ID

When the application is in edit mode, the user can assign the current parameter to any of the three Chameleon panel potentiometers (pot), by holding pressed the shift key when turning left or right the selected pot. When this is done, the LED located above the used pot will be turned on indicating that this has an assigned parameter. If the user turn left or right one the assigned knobs, the application changes to the assign mode.

When the edit mode begins, the edit LED is turned on. If any of the parameters of the current preset has been modified, then this LED will flash to notice this fact.

To exit the edit mode, the user has to press the edit key. If the current preset has been modified, the application change to the save mode. If not, then the application is put in the preset mode again.

- Save mode (*CMonoSynthPanelModeSave*): This mode is used to ask the user for information to save the current modified preset. First, the display shows a message asking the user to confirm the operation. The user can press the edit key to accept the current displayed answer, press the value up and down keys to change the answer to Y or N respectively, or press the shift key to cancel the operation and return to the preset mode. While waiting for an answer, the shift and edit LEDs will flash alternatively to inform the different possibilities.

```
Preset modified
Store it? [Y]
```

If the user's answer to the previous question was Y, then the application asks the user for the target preset number to be overwritten, showing the current one. The user can use the encoder or the value up and down keys to change this value. As in the previous step, the edit key is used to accept the current setting, and the shift key to cancel the process returning to the preset mode.

```
Preset number?
[001]
```

If the user pressed the edit key in the previous step, then the applications asks the user for a new preset name, showing the current one. The user can use the encoder or the value up and down keys to change the current flashing letter (cursor). The param up and down keys can be used to advance or to move back the cursor. As in the previous step, the edit key is used to accept the current setting, and the shift key to cancel the process returning to the preset mode. Pressing the edit key will go also to the preset mode but storing the changes made in non volatile memory.

```
Preset name?
[Fat Bassey    ]
```

- **Assign mode** (*CMonoSynthPanelModeAssign*): This mode is used to inform the user that an assigned knob has been moved and to display the parameter name and value changed. This information is displayed during approximately 1.5 seconds and then the application change to the previous mode again.

```
 Filter Cutoff
     [ 84]
```

- **Midi mode** (*CMonoSynthPanelModeMidi*): This mode displays the MIDI Dump menu to allow the user to send a bank or preset MIDI system exclusive dump to the Chameleon's MIDI out connector.

```
MIDI Dump:
[Current Preset]
```

The user can press the value up and down keys to change the current option (Full Bank or Current Preset), the edit key send the MIDI dump, or the shift, group up or group down keys to cancel the process and to return to the preset mode. In this mode, the edit and shift LEDs will flash alternatively to notice the available options.

DSP Handler

The **DSP handler** class (*CDsp*) is derived also from the *CMessageHandler* and therefore it is implemented as a RTEMS task runnning in parallel with the other tasks. It contains internally a reference to the DSP driver and together with the assembler host code implements a set of enqueued commands (accesible as messages) which allow to:

- Write a block of adjacent words in P, X or Y memory.
- Write a single word in any P, X or Y memory location.
- Internally copy P memory blocks of code.
- Fill blocks of either X or Y memory data with a specified value.

The class *CDsp* is used by *CMonoSynthRenderer* directly to update the parameter changes when the *CMonoSynthEngine* class decides. The *CMonoSynthRenderer* is an alone class which contains privately all parameters involved in the synthesizer (all derived from *CParameter*) and a set of functions called by *CMonoSynthEngine* to update the parameter values in the DSP (Update), implement voice allocation (DoMidiKeyOn, DoMidiKeyOff), and other MIDI related work (DoMidiPitchWheel).

Internally, the *CMonoSynthRenderer* class knows how the DSP assembler code is made and it uses the constants created when assembling the DSP code (DSPP_, DSPX_, DSPY_, DSPL_, DSPK_) to modify the DSP memory making calls to the private *CDsp* class. To speed up the computation of the DSP coefficients associated to parameters, several pre-filled internal tables are used (note frequencies, logarithms...)

For example, when the Master Volume parameter must to be updated, the *CMonoSynthEngine* calls to *CMonoSynthRenderer::Update*, passing the constant kGlobalMasterVolume as argument. This function calls to *CDsp::SendMsgWriteWordX* using the constant DSPX_MasterVolume (which is generated when assembling the DSP code, because there is a label in X memory called MasterVolume) and the value returned by GetAttenuation_dB with the current Master Volume parameter value as arguments. When the DSP handler receives the message generated when SendMsgWriteWordX was called, it directs the DSP using the DSP driver to write the specified word in the specified DSP memory address.

MIDI Handler

The MIDI handler class (*CMonoSynthEngine*) is derived from the class *CEngine*, which is in turn derived from *CMessageHandler* and therefore it is implemented as a RTEMS task runnning in parallel with the other tasks. *CEngine* contains a private reference to a MidiShare task used to send/receive MIDI data to/from the external MIDI ports. The MIDI data is received from the local message queue using the standard method proposed in the *CMessageHandler* class (OnMessage). The class supports the channelized MIDI messages Note On, Note Off, Pitch Bend, Controller Change and Program Change, and also the following system exclusive messages: stored preset dump (override flash preset), edited preset dump (override only edit buffer), stored preset request, edited preset request and stored bank request.

CMonoSynthEngine uses several tables to map the available parameters to the configuration data format (m_tableGlobalParameters), the preset data format (m_tablePresetParameters) the received MIDI controllers (m_tableMidiCtrlParameters) and the sent MIDI controllers when panel knobs are moved (m_tablePot2MidiCtrl).

The parameter changes are directed to the private *CMonoSynthRenderer* object, as described in the previous section.

The *CMonoSynthEngine* class contains also a reference to another MidiShare task: the player. This is used to show how to play a standard MIDI file using the sequencer supplied in the Player library. The MIDI file is included in the project using the utility called BIN2HEADER supplied with the Chameleon SDK.

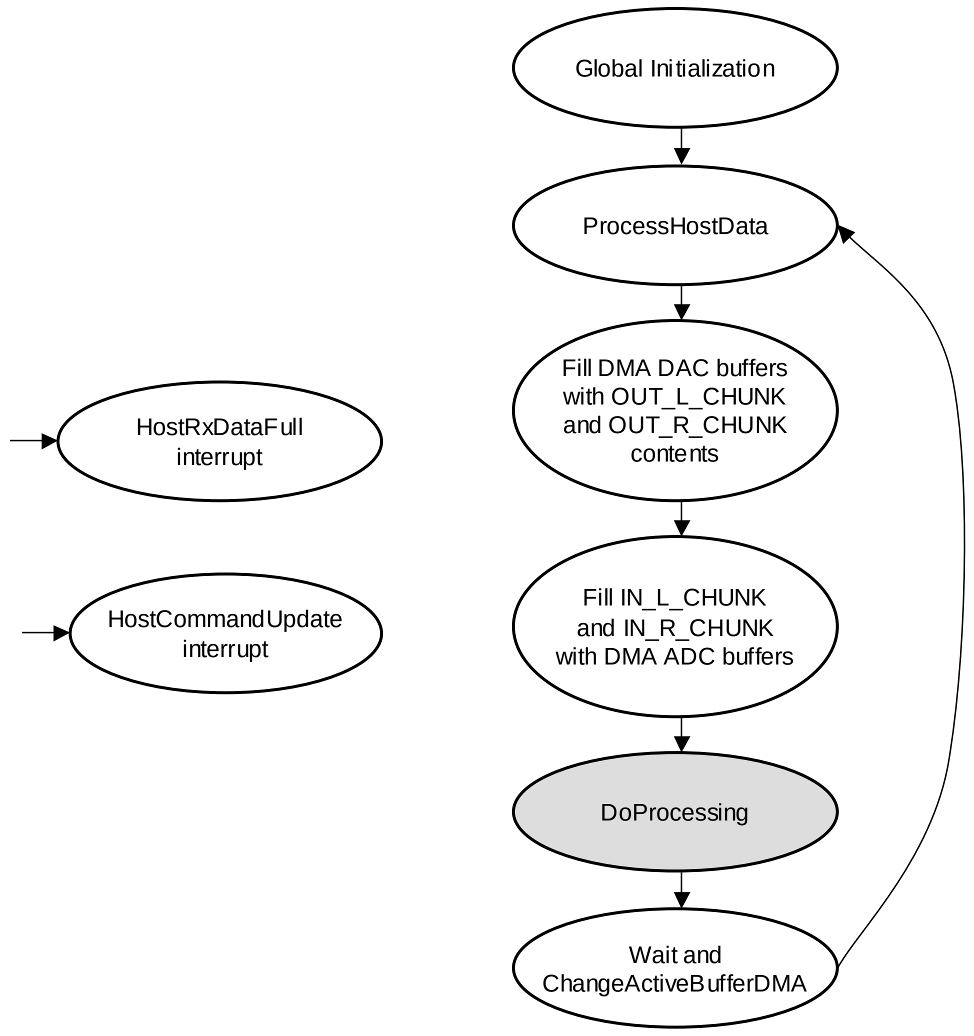
This class has also a reference to the flash driver used to load and save the preset and configuration data in non volatile memory.

DSP Side

The DSP part of the application is composed by a main assembler file (main.asm) which includes the rest of the files involved. This file contains mainly the initialization code and the main loop, inside which is included the file prg.asm containing the real synthesizer processing code using re-usable modules.

The communication with the ColdFire is made by using host interrupts: each time the ColdFire writes data to the DSP, the HostRxDataFull interrupt is asserted and the ISR saves the data consecutively into an internal HostBuffer. When the ColdFire writes a host command 0x32 (HostCommandUpdate), the DSP disables reception of data in the host port. Each time the main loop is executed, the ProcessHostData tests if the data reception is disabled, and if so, then it decodes the data sent by the ColdFire (which is located in the HostBuffer) and updates DSP memory locations. This data is coded using a custom protocol designed to implement the messages described in the *CDsp* class of the ColdFire.

Figure 2
Flow diagram of the DSP assembler code


Global Initialization
ProcessHostData
Fill DMA DAC buffers with OUT_L_CHUNK and OUT_R_CHUNK contents
HostRxDataFull interrupt
HostCommandUpdate interrupt
Fill IN_L_CHUNK and IN_R_CHUNK with DMA ADC buffers
DoProcessing
Wait and ChangeActiveBufferDMA

The initialization code configures the DSP to use the DMA block mode for making audio transfers using a double buffer method, in wich while the core uses a buffer to transfer audio using the DMA, the program is using the other buffer for processing; then, when the program finish the actual buffer processing, it waits for DMA completion and then swaps the buffer usage.

The processing method used is therefore block based, using blocks of CHUNK_SIZE samples. The processing code uses a predefined structure to allow a modular-like programming framework, which follow the next guidelines:

- Each processing module receives arguments in consecutive memory addresses pointed by R_X and R_Y (defined as R0 and R4 respectively int file main.asm).

- The inputs and outputs are called CHUNKS. Each CHUNK contains CHUNK_SIZE samples. The module must to process exactly CHUNK_SIZE samples from each input and generates CHUNK_SIZE samples into each output.

- Each module can use any number of CHUNKS as input or output. The same CHUNKS can be used as input or output by the same or different modules.

- Each module can use other fixed memory zones to store tables of any size (wavetables, delay lines...)

For example, this block of code implements a typical two inputs mixer using the proposed framework:

```
;*********************************************************
;
; Defining the input and output chunks
;*********************************************************
;
      org    X:

In1:  CHUNK

      org    Y:

In2:  CHUNK
Out:  CHUNK

;*********************************************************
;
; Defining module arguments
;*********************************************************
;
      org    X:

ModulesDataX:
      XMEM  Mixer_In1Addr,In1
      YMEM  Mizer_OutAddr,Out

      org    Y:

ModulesDataY:
      YMEM  Mixer_In2Addr,In2
```

```
;*******************************************************
; Module code contents: when calling this piece of code
; the R_X must point to ModulesDataX and R_Y to ModulesDataY
;*******************************************************
      org    P:

      MOVE   X:(R_X)+,R1      ; (X)   = In1Addr (X)
      MOVE   Y:(R_Y)+,R5      ; (Y)   = In2Addr (Y)
      MOVE   X:(R_X)+,R6      ; (X+1) = OutAddr (Y)

      BEGIN_LOOP
      ;-------------------------------------------------
        MOVE X:(R1)+,X0       Y:(R5)+,A
        ADD  X0,A
        NOP
        MOVE A,Y:(R6)+
      ;-------------------------------------------------
      END_LOOP
```

The symbols CHUNK, XMEM, YMEM, BEGIN_LOOP and END_LOOP are defined in main.asm to understand easily the code. In this example, we use three different chunks of samples: two inputs and an output. One of the input chunks is located in X memory; the other input is in Y memory; and the output chunk is in Y memory.

The module arguments are stored in consecutive memory beginning from R_X and R_Y:

(R_X + 0) = In1Addr
(R_X + 1) = OutAddr

(R_Y + 0) = In2Addr

The first argument in the XMEM and YMEM macros (both defined in the file main.asm simply as a define constant directive, but used to distinguish X and Y memory arguments) is a label used to reference the argument from the ColdFire using the constants DSPX_ and DSPY_

The code first load the arguments (here pointers) in registers and execute CHUNK_SIZE times the loop, inside which the code will load the contents of the input chunks sample by sample, will add both, and will save the result in the output chunk.

In the MonoSynth application the following modules have been created (which are not optimized for speed nor quality and they are only provided as examples):

- **osc** : It implements an interpolating wavetable oscillator with fixed amplitude and frequency. The example is supplied with four different wavetables: a sine wave, a triangle wave, a sawtooth wave and a square wave. It has been also included a Microsoft Excel document (doc/wavetables.xls) to show how to make additional wavetables. **NOTE**: the oscillator is not bandlimited; thus, the output signal can be aliased if the wavetable contains abrupt changes (as for example the sawtooth and the square wave provided).

- **noise** : This module generates pseudo-white noise using the linear congruential method.

- **mixer** : The mixer adds four inputs and fills an output, scaling two of them before by using the provided scale constants in the arguments.

- **env** : This module computes piecewise-exponential envelope segments specified by duration/target/rate tables in memory.

- **scale** : This module simply scales the input CHUNK using the provided constant.

- **addk** : This is other simple module which adds a constant value to the input CHUNK.

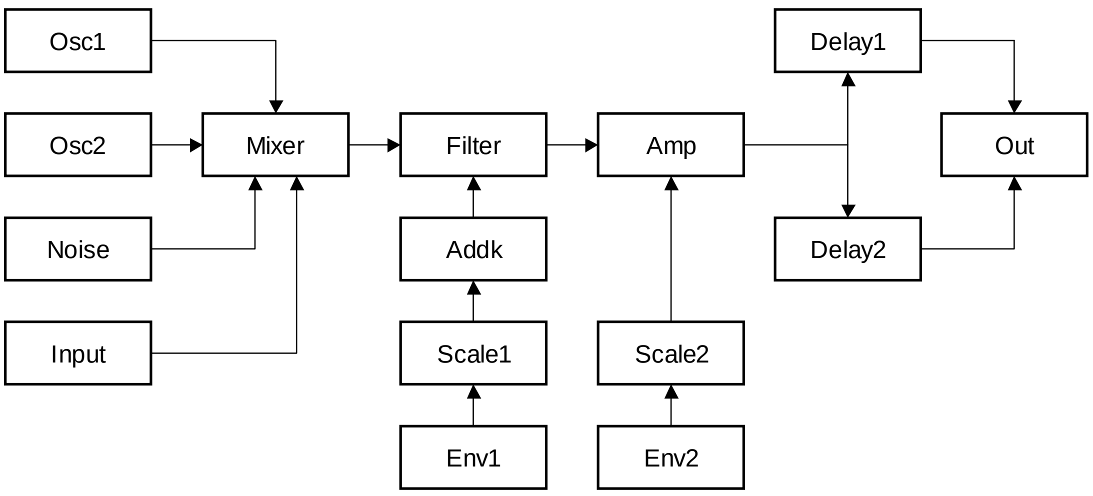
- **filter** : The filter module implements a typical 24dB lowpass resonant VCF (voltage controlled filter), with a control input CHUNK used to modulate the filter cutoff. **NOTE**: the filter is not well adjusted, and when the cutoff input is close to zero, the filter tend to be unstable, generating clicks at output.

- **amp** : This module implements a typical VCA (voltage controled amplifier), with a control input CHUNK used to modulate the amplifier gain.

- **delay** : This modules implements an interpollating fractional delay line with feedback, used in the example to simulate an echo effect.

- **out** : The out module is used to accumulate the input CHUNKs to the output buffers which will be send to the D/A converter next time.

Figure 3
monosynth modular structure



Using these modules, the file prg.asm defines a typical analog synthesizer structure with two oscillators, a noise generator and an input source mixed and sent to the input of a lowpass filter, which cutoff is modulated using an envelope generator, and whose output is sent to an amplifier with gain modulated by another envelope generator. The output of the amplifier is processed by two delay modules to simulate a stereo echo effect and the outputs of the delays are sent to the output module and therefore to the outside world.

The ColdFire (*CMonoSynthRenderer*) uses the labels defined in the arguments declarations of the modules to make changes according to the MIDI input and parameter changes made to re-create the meaning of typical synthesizer parameters.