Signal Essence MMFx Data Sheet

Revisions

Date	Comment
2015.10.27	Clarified relationship between RX's rout and TX's refIn (RY)
2015.12.9	added 48khz example; edited loop example to show 10msec interval (RY)
2015.12.18	fixed 48khz example; added section on rout format and header files (RY)
2015.12.22	updated integration checklist (CC)
2016.03.14	In block diagram, added final gain stage to Tx path
2016.04.20	Added clarification about sediag
2016.09.12	Added section about querying sediag for location search
2016.09.14	Added policy algorithm discussion
2016.09.29	Added instructions for customizing policy algorithm Moved and updated integration steps Added discussion of ref_in bulk delay and configuration
2017.05.12	Updated MMIfInit
2017.08.23, 2017.09.11	Added sections on output gain and monitoring power
2017.11.03	Spatial Filter Bypass Enabling/Disabling specific microphones
2017.11.27	Forcing location search
2018.07.12	Updated output gain getter/setter
2018.07.25	enable/disable SENR

MMFx PRODUCT OVERVIEW

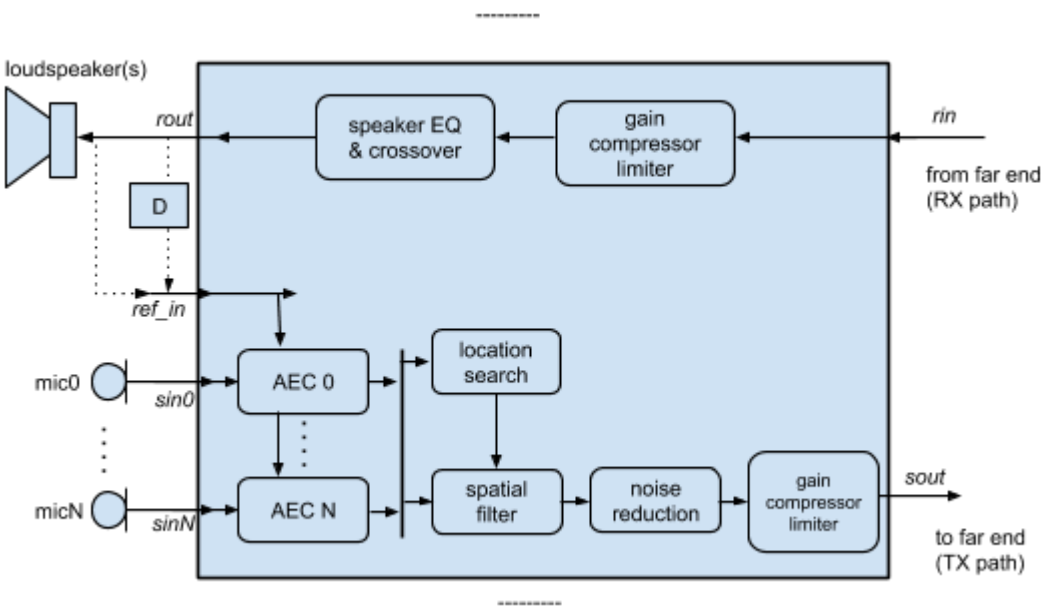
The Signal Essence MMFx (Multi-Microphone Effects) Software provides high-quality, full-duplex speakerphone audio functionality to your product by implementing acoustic echo cancellation, multi-microphone location search and beamforming, and noise reduction. On the loudspeaker side, the MMFx dynamic gain control algorithm works in tandem with the microphone processing to provide the best full-duplex audio performance. Signal Essence customizes MMFx to make optimal use of your product's loudspeaker and microphone configuration.

Key Features

Location search and spatial filtering: applies multi-microphone beamforming to improve microphone pickup
Acoustic Echo Cancellation (AEC): cancels echo to allow full-duplex communication
Noise reduction (NR): suppresses residual echo and eliminates non-speech noise
Speaker gain control and EQ: optimizes your loudspeaker output
Implemented in Standard C: Port MMFx to accommodate nearly any compute platform
Simplified Integration: A simple API and detailed diagnostics simplify software/hardware integration

MMFx ARCHITECTURE AND OPERATION

The signal processing paths for MMFx are shown in the figure below. By convention, MMFx represents the near-end of a two party connection; signals received by MMFx from the far-end are handled by the receive path (RX), and signals transmitted to the far-end are handled by the transmit path (TX).



MMFx Architecture

On the TX path, MMFx accepts signals from each microphone (sin0..sinN), along with a "reference signal" (ref_in) representing the signal sent to the loudspeaker. An acoustic echo canceller (AEC) cancels echoes originating from the loudspeaker; each microphone is cancelled individually. The duration of the expected echo reverberation time (the "tail length") is configurable to handle various acoustic environments and CPU limitations. The AECs are frequency-domain adaptive filters with backup banks, assuring fast channel model convergence and excellent stability. ERLE values (echo return loss enhancement: a measure of echo cancellation performance) typically range from 15 to 25 dB.

Following echo cancellation, MMFx spatially locates the near-end talker. MMFx then performs spatial filtering by combining multiple microphone inputs to steer the microphone pickup pattern at the near-end talker. Spatial filtering improves microphone directionality, thereby reducing reverberation as well as filtering out noise from other directions. In some cases, MMFx can also simultaneously isolate a noise source and remove it from the near-talker.

Next, the MMFx noise reduction algorithm processes the near-talker, loudspeaker signal, and noise reference to suppress any remaining echo as well as reduce non-speech noise. Noise reduction parameters are highly configurable, and can be tuned, for example, to work with speech recognition systems. The resulting clean speech signal is sout.

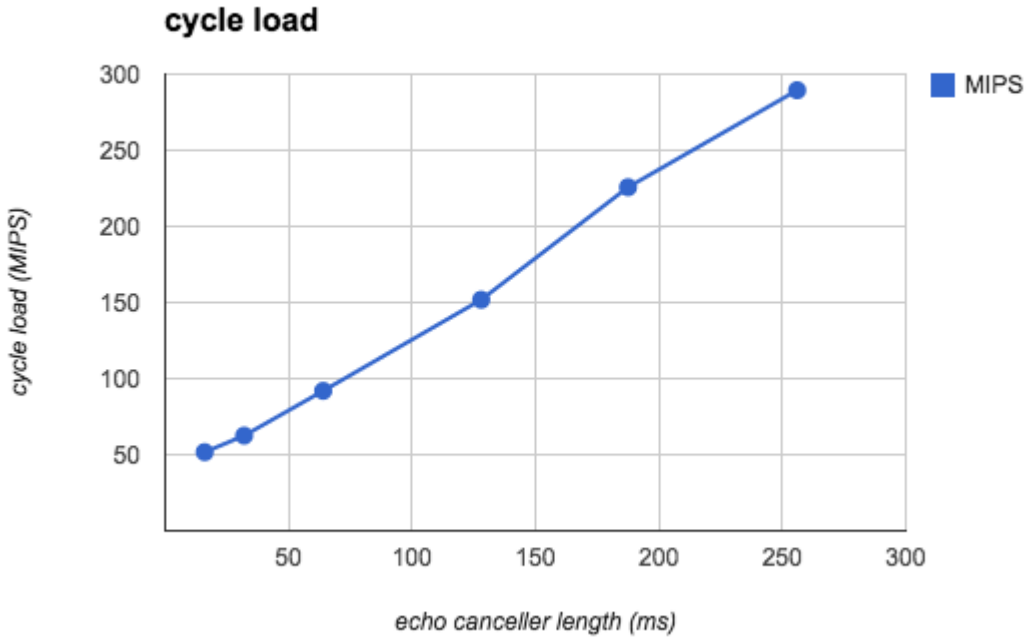
On the RX side, MMFx receives a signal from the far end (rin). MMFx applies dynamic gain control and equalization to the signal destined for the loudspeaker (rout). Systems that require louder loudspeakers often benefit from using a 2 or 3-way loudspeaker with a woofer/tweeter combination. MMFx handles the crossover, which simplifies system design. The receive and

transmit processing work together to optimize sound quality; for example, the transmit side can dynamically reduce the loudspeaker gain in order to improve full-duplex behavior.

PROCESSOR AND MEMORY REQUIREMENTS

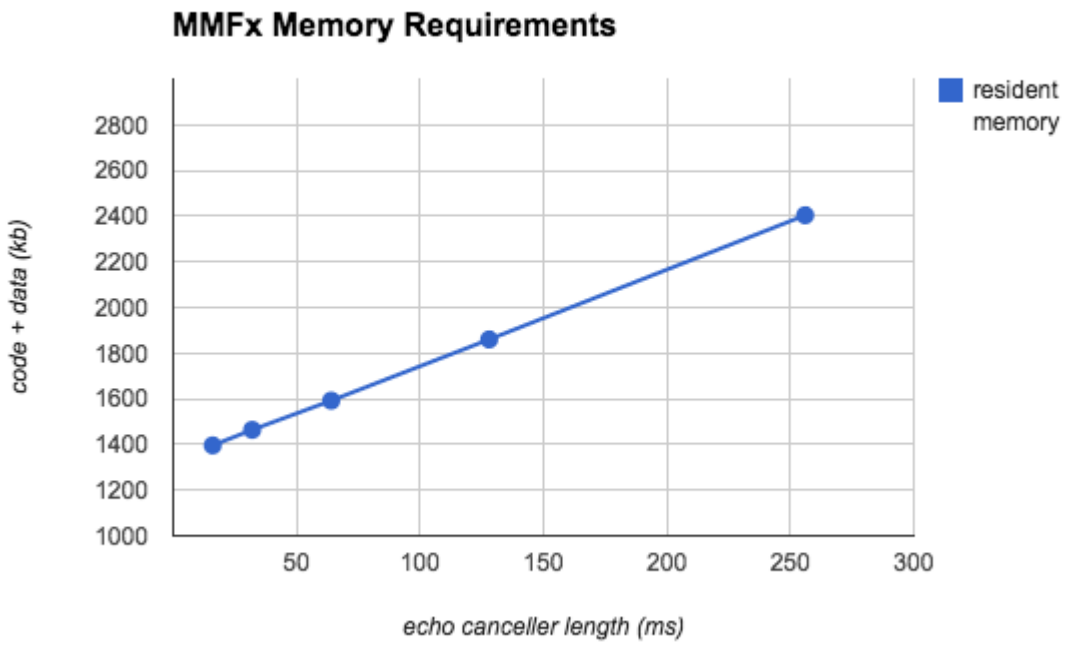
MMFx is implemented in floating-point ANSI C, so it can be ported to any platform with floating point hardware and a C compiler. Some functions may need to be hand optimized for each target CPU. We currently have optimized implementations for Intel (32 and 64-bit), ARM NEON, and the TI TMS320C67xx.

CPU utilization is primarily dependent on the echo canceller tail length, and to a lesser degree, the number of microphones, number of spatial solutions, and the degree of CPU-specific optimization. For example, the following chart shows cycle load versus the echo canceller tail length on an ARM Cortex-A8; MMFx is configured to process 10 ms sample blocks and one microphone.



Cycle Load (Cortex A8) as a Function of Echo Canceller Length

MMFx explicitly avoids dynamic memory allocation on the heap; memory allocations occur either on the stack or in a fixed-size memory pool. The amount of memory required depends primarily on the echo canceller tail length, as well as the number of microphones and spatial solutions. The chart below illustrates memory usage on an ARM Cortex-A8 as a function of echo canceller length; MMFx is configured with 6 microphones.



Memory Utilization as a Function of Echo Canceller Length

SOFTWARE INTEGRATION

Signal Essence delivers MMFx as object code. For each customer, we also develop a customized software interface layer, called MMIF, which lies between your application code and MMFx. MMIF performs customer-specific MMFx configuration and exposes a simple API for easy integration. MMIF is delivered as source code (ANSI C), which you are free to modify.

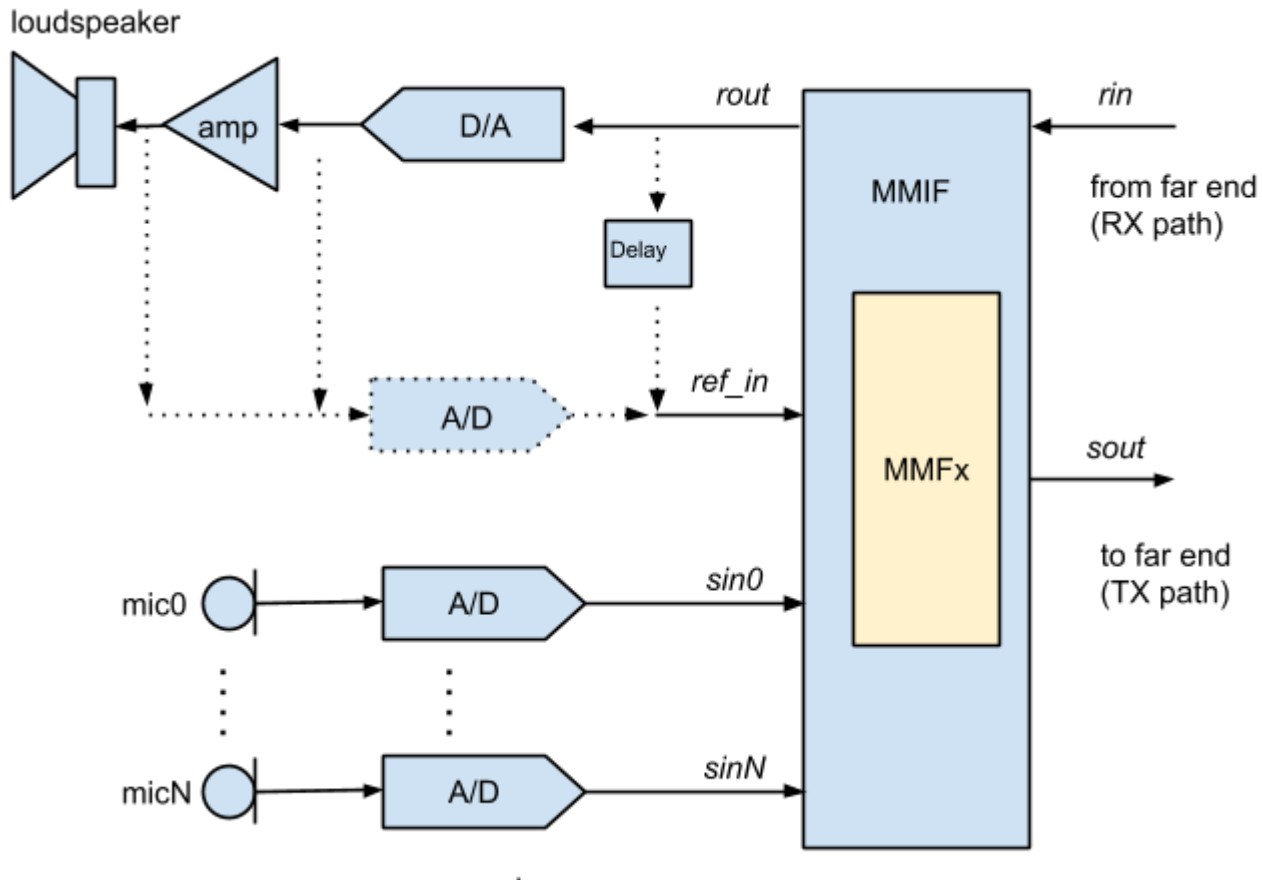
Your software must execute MMFx processing every 10 ms. MMFx audio samples are specified as 16-bit integer values, [-32768.0 .. 32767.0], sampled at 8 or 16 kHz.

Input and output signals must take the form of 10 msec sample blocks, 160 samples per block (or 80 samples per block for 8kHz sample rate). Sample blocks are un-interleaved, i.e. two microphone inputs are passed to MMFx as follows: {mic0[0..159], mic1[0..159]}.

The figure below illustrates how MMIF interfaces between your software and MMFx. For reference, here is a summary of the input and output signals (see MMFx Architecture and Operation):

- `rin`: RX path input samples; audio from the far end; possibly multi-channel

- rout: RX path output samples, possibly multi-channel; audio destined for the loudspeaker
- ref_in: a delayed version of rout. This signal should be derived from rout as close to the loudspeaker as possible. Alternate paths for ref_in are shown with dotted lines in the diagram; ref_in is further discussed below (Providing a Reference Signal (ref_in) to MMFx) . May be multichannel.
- sin0..sinN: per-microphone audio sample blocks, one block per microphone, deinterleaved samples
- sout: TX path output; audio to the far end



MMFx, MMIF, and I/O Signals

The main MMIF function prototypes are in mmif.h:

```
void MMIfInit( float32 additionalRefDelaySec, void *pArgs);

void MMIfProcessMicrophones(int16 *refInPtr
                            int16 *sinPtr,
                           int16 *soutPtr);

void RcvIfProcessReceivePath(int16 *rinPtr,
```

```
                                int16 *routPtr);

void MMIfDestroy(void);
```

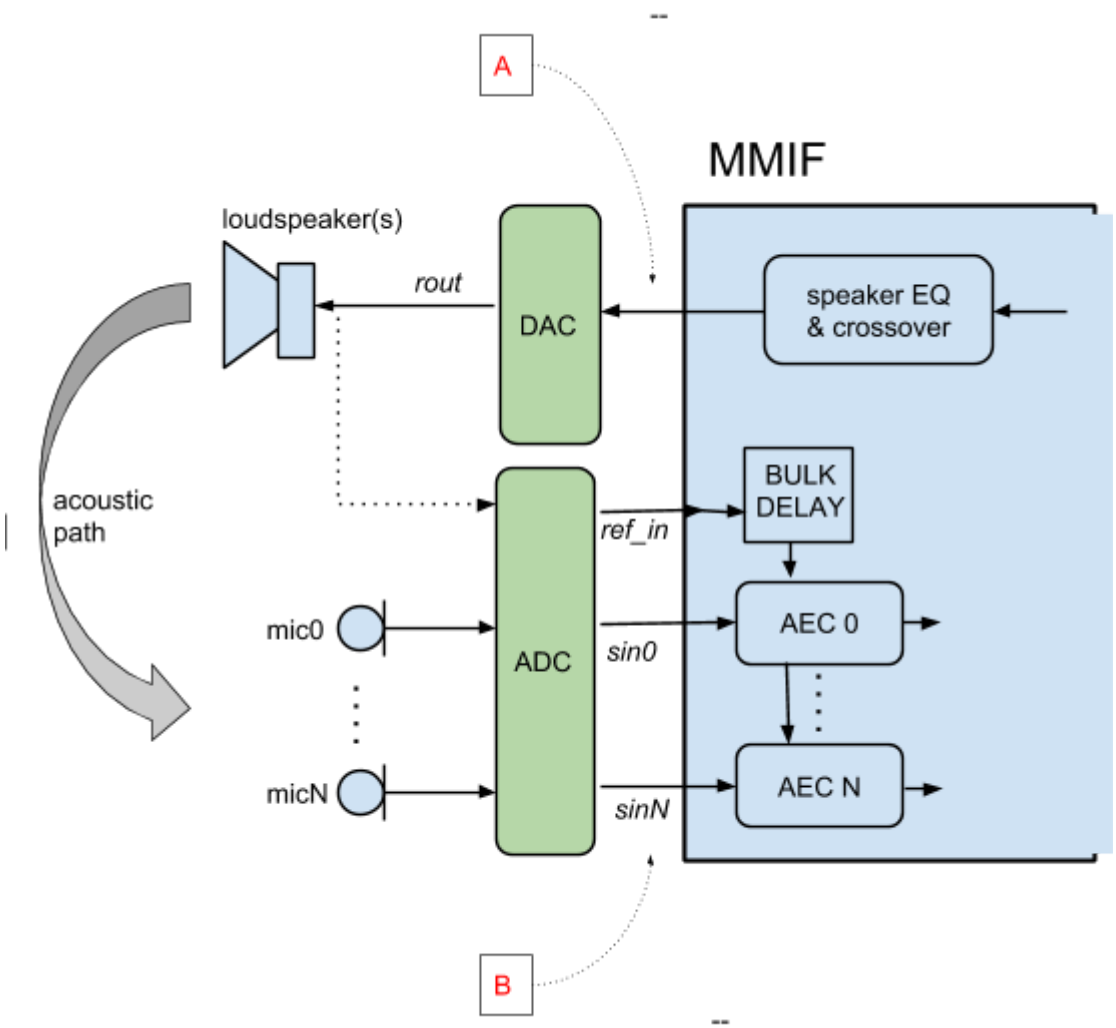
Header Files

Your application code must include 2 header files:

```
#include "mmif.h"          // general MMIF definitions, located in /se_lib_public
#include "mmif_proj.h"     // project-specific definitions
```

Specifying the Bulk Delay in MMIfInit()

The bulk delay refers to the time delay between when rout enters the DAC (at A) and the delayed echo re-enters sin at B (see the diagram below), i.e. bulk delay = t(B) - t(A). A delay is applied to ref_in so that ref_in arrives at the Acoustic Echo Cancellers (AECs) approximately 5 ms prior to the echo.



The bulk delay is specified when your audio application initializes MMIF; the units are in seconds.

For example, assume that an impulse at rout incurs 255 ms of delay between points A and B. Ref_in should be delayed by 255 ms - 5 ms = 250 ms, so that the impulse arrives at the AECs slightly ahead of the impulse echoes.

In your audio application, the bulk delay is specified during initialization:

```
MMIfInit(250.0e-3, NULL); // delay ref_in by 250 msec
```

Providing a Reference Signal (ref_in) to MMFx

In order for MMFx to effectively cancel echoes, the ref_in input signal must be a faithful representation of the signal going out to the loudspeaker. Assuming that the system has deterministic latency and the power amplifier is linear and time invariant, then ref_in can be simply a delayed version of rout (the signal headed to the DAC/loudspeaker).

There are at least 2 situations where the ref_in signal cannot be derived directly from rout in software:

- The software/OS is incapable of providing a deterministic latency upon startup
- Any nonlinear processing is applied to the loudspeaker signal; for example if a "smart" power amplifier or DAC is used that manage the loudspeaker output gain or frequency response.

In this case, you will need to allocate an additional ADC converter channel to sample the signal that goes out to the loudspeaker. This creates a nicely time-aligned version of ref_in to come in with the microphone signals. If only the DAC is nonlinear, the signal can be sampled between the DAC and PA. If the PA is nonlinear (i.e. some PAs will turn gain down when input voltage is low), then it must be sampled after the PA. This can be tricky if your amplifier is class-D.

Format of Rout

The Receive Path output, rout, is intended to 1..N speakers (where N is determined by your project). Typically the number of channels will be one, for a single-speaker output, or three, representing the monitor, woofer, and tweeter signals. When rout is multichannel, the "monitor" channel is the audio output prior to splitting into woofer and tweeter signals. The mapping of subsequent channels to woofer, tweeter, etc. is determined per-project.

For example, say your project's receive path output works at 48 kHz and will drive both woofer and tweeter. Rout will contains monitor, woofer, and tweeter signals. When calling the Receive Path function:

```
void RcvIfProcessReceivePath(const int16 *rinPtr,
                             int16 *routPtr )
```

you must pass routPtr = a buffer large enough to hold 3 * 480 samples (3 channels * 480 samples/block). Upon calling RcvIfProcessReceivePath(), RoutPtr will receive 3 sample blocks with the format:

```
block 0 (0..479)   : rout monitor channel
block 1 (480..959) : woofer output
block 2 (960..1439): tweeter output
```

Example MMIF Usage (16 kHz Sample Rate)

A typical usage of the MMIF API is shown below. Assume that:

- the sample rate throughout the system is 16 kHz
- rinPtr contains 160 samples of rin (RX path input samples); 160 samples at fs=16 kHz is 10 msec.
- routPtr is a 160 sample buffer which will hold rout (processed audio destined for the loudspeaker); in this example, rout is single-channel.
- refInPtr contains 160 samples of the reference signal for the echo canceller (ref_in; a delayed version of rout)
- the array uses 6 microphones, so sinPtr contains 6 * 160 samples of microphone inputs
- soutPtr is a 160 sample buffer which will hold sout (processed audio output)

Within the main loop, the MMIF receive and transmit functions need to execute every 10 msec.

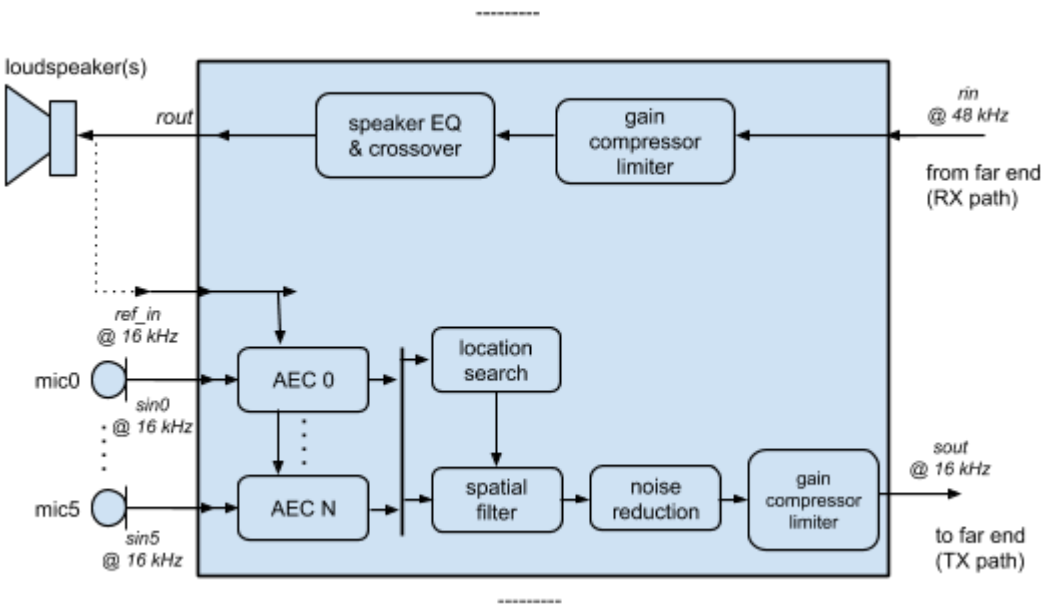
```
int16 rinPtr[160], routPtr[160], refInPtr[160], sinPtr[6 * 160], soutPtr[160];

// initialize MMFx
// and specify additional ref_in delay
MMIFInit(0, NULL);

//
// do receive and transmit processing
while (every 10 msec)
{
      RcvIfProcessReceivePath(rinPtr,    // in
                              routPtr);  // out
      // routPtr is played out through the loudspeaker

      // in this example, refin and rout are identical
      memmove(refInPtr, routPtr, 160 * sizeof(int16));

      MMIfProcessMicrophones  (refInPtr,  // in
                               sinPtr,    // in
                               soutPtr);  // out
}

MMIFDestroy();
```

Notes:

- The RX function, RcvIfProcessReceivePath(), should be called before the TX function.
- You may also call the RX and TX functions in separate threads. In this case, you are responsible for calling the RX and TX functions synchronously/lockstep, and in a thread-safe manner. You must also ensure that the RX function is called first.

Example MMIF Usage (Mixed Sample Rates)

For music applications, we recommend using a 48 kHz sample rate. Signal Essence can also configure MMFX to work with mixed input and output sample rates, e.g. the entire RX path works at fs=48 kHz, while the TX path receives fs=16 kHz inputs and outputs fs=16 kHz.



Assume that:

- rinPtr contains 480 samples of rin (audio destined for the loudspeaker); 480 samples at 48 kHz is 10 msec. Rin is single-channel.
- routPtr is a 3 * 480 sample buffer which will hold rout (processed audio destined for the loudspeaker); in this case, rout contains three channels.
- refInPtr contains 480 samples of the reference signal for the echo canceller (ref_in).
- sinPtr contains 6 * 480 samples of the microphone inputs. Sin is 6-channel--one channel per microphone.
- soutPtr is a 160 sample buffer which will hold sout (processed audio output).

Within the main loop, the MMIF receive and transmit functions need to execute every 10 msec.

```
int16 rinPtr[480], routPtr[3 * 480], refInPtr[480], sinPtr[6 * 480], soutPtr[160];
```

```
// initialize MMFx
// and specify additional ref_in delay
MMIFInit(0, NULL);

//
// do receive and transmit processing
while (every 10 msec)
{
       RcvIfProcessReceivePath(rinPtr,    // in 48 kHz
                               routPtr);  // out 48 kHz
       // routPtr is played out through the loudspeaker



       // in this example, refInPtr comes from the codec
       MMIfProcessMicrophones  (refInPtr,  // in 48 kHz
                                   sinPtr,    // in 48 kHz
                                   soutPtr);  // out 16 kHz
}

MMIFDestroy();
```

Diagnostic Interface (sediag)

MMFx has a built-in diagnostic interface, called sediag, which publishes a large number of variables representing operating parameters and metrics. You access these variables during runtime for fine-grained control and monitoring.

Diagnostic entries are indexed by string labels; for example, in order to set the transmit path gain, we get the ID number mapped to the diagnostic “mmfx_gsout_q10” and modify the variable with a setter function:

```
int id = SEDiagGetIndex("mmfx_gsout_q10"); // get id mapped to string label
SEDiagSetInt16(id, gainValue);             // set value
```

Using the sediag reflection functions, you can get a complete list of diagnostic string labels, along with associated ID, type, length, and description:

```
int          SEDiagGetNumDiags(void);
int          SEDiagGetIndex   (const char *name);
int          SEDiagGetLen     (int index);
SEDiagType_t SEDiagGetType    (int index);
SEDiagRW_t   SEDiagGetRW      (int index);
const char  *SEDiagGetName    (int index);
const char  *SEDiagGetHelp    (int index);
```

For initial integration and debugging, Signal Essence requires that customers implement a method for setting and getting diagnostics in their software during runtime. A convenient method for exposing the interface is to implement a simple web server in your application; we

use the miniHTTP library. Signal Essence can supply example HTML and Javascript for this purpose.

Here is a list of getters and setters for diagnostic entries:

```
// getters
int16   SEDiagGetInt16        (int   index);
uint16  SEDiagGetUInt16       (int   index);
int32   SEDiagGetInt32        (int   index);
uint32  SEDiagGetUInt32       (int   index);
int64   SEDiagGetInt64        (int   index);
uint64  SEDiagGetUInt64       (int   index);
float32 SEDiagGetFloat32        (int   index);
int16  *SEDiagGetInt16Array   (int   index);
int16  *SEDiagGetInt16ArrayI  (int   index, int i);
uint16 *SEDiagGetUInt16Array  (int   index);
int32  *SEDiagGetInt32Array   (int   index);
uint32 *SEDiagGetUInt32Array  (int   index);
int64  *SEDiagGetInt64Array   (int   index);
uint64 *SEDiagGetUInt64Array  (int   index);
float  *SEDiagGetFloat32Array (int   index);

// setters
void   SEDiagSetInt16        (int   index, int16  value);
void   SEDiagSetUInt16       (int   index, uint16 value);
void   SEDiagSetInt32        (int   index, int32  value);
void   SEDiagSetUInt32       (int   index, uint32 value);
void   SEDiagSetInt64        (int   index, int64  value);
void   SEDiagSetUInt64       (int   index, uint64 value);
void   SEDiagSetFloat32      (int   index, float32  value);
void   SEDiagSetInt16N       (int   index, int16   value, int n);
void   SEDiagSetUInt16N      (int   index, uint16  value, int n);
void   SEDiagSetInt32N       (int   index, int32   value, int n);
void   SEDiagSetUInt32N      (int   index, uint32  value, int n);
void   SEDiagSetInt64N       (int   index, int64   value, int n);
void   SEDiagSetUInt64N      (int   index, uint64  value, int n);
void   SEDiagSetFloat32N     (int   index, float32 value, int n);
```

Querying sediag for Location Search Results

The default behavior of MMFx's direction-of-arrival algorithm (i.e. "location search") is to scan over a set of pre-determined directions every 10 msec and return a power metric (expressed as a confidence value) for each direction. The direction with the highest confidence should represent the direction of the strongest signal.

In reality, the direction of highest confidence can jump quite a bit because the beamforming array is sensitive to noise and echoes. We have found that a more stable indicator for

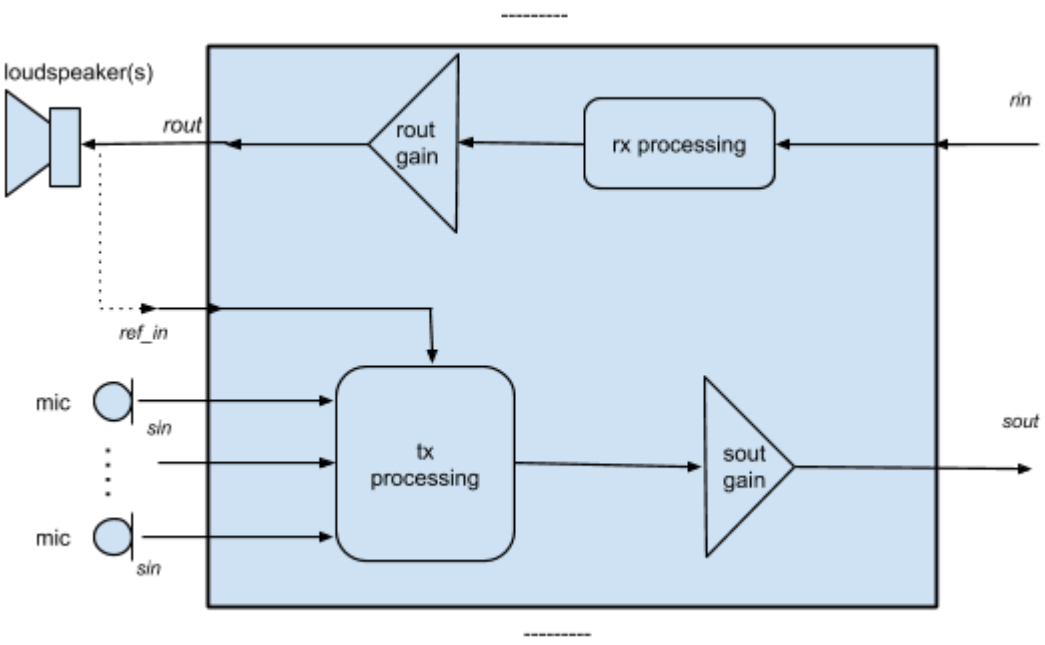
direction-of-arrival can be computed by forming a moving histogram of the highest confidence direction over a longer interval, then picking the direction bin with the most hits. In effect, the algorithm picks the direction with the most votes. This algorithm is built into MMFx.

During runtime, you can query sediag for either the "raw" result or the histogrammed result. Use the following sediag strings to tune the parameters and read the results:

Name	Type	Length	RW	Description
fdsw_current_winner_beam	uint16	1	R	The index of the "winning" direction (the direction according to the histogram algorithm)
fdsw_len_delay_line	uint16	1	RW	The duration over which to collect the histogram, specified in multples of sample blocks (typically 1 sample block = 10 msec)
fdsearch_num_beams_to_search	uint16	1	R	The number of search beams (directions)
fdsearch_beam_xyz_nnn, e.g. fdsearch_beam_xyz_001	int16	3	R	The direction, expressed as a (x, y, z) vector in cm. nnn is an index, pre-padded with zeroes, which varies from 000..(N-1), where N is the result of querying fdsearch_num_beams_to_search.
fdearch_best_beam_index	uint16	1	R	the index of the beam with the highest confidence for the current sample block
fdsearch_best_beam_confidence	int16	1	R	the confidence of the "best" beam, expressed as a Q15 metric (0..32767)
fdsearch_current_confidence	float32	# of search beams	R	array of confidence metrics for each beam

Setting Output Gain (Volume Control)

Both rout (the loudspeaker) and sout (the microphone-side output) have separate gain controls. You should always use the MMFx gain controls, rather than applying your own gain stage, because MMFx applies processing to limit signal clipping and optimize output power.



Rout **Gain (Loudspeaker)**

You can set the rout gain using two methods:

- sediag: the label "sercv_gain_per_chan_q10" maps to a read/write array of integers; each integer represents the gain for one receive-path channel. For most customers, there is only one receive-path channel.
  The integer represents gain as a 16-bit integer [0..32767], where the gain g is represented as a fixed-point Q10 value:

  $$g = \frac{q10}{2^{10}}$$

  Thus q10 = 1024 represents unity gain (1024/1024 = 1), and q10 = 32767 represents 31.9 (= 32767/1024).
- C function:
  `mmif.h::MMIfGetRoutGainQ10(void)`
  returns the Q10 gain of rout channel 0.

  `mmif.h::MMIfSetRoutGainQ10(int16 gain_q10)`
  sets the gain of all rout channels to the Q10 value.

Sout **Gain (Microphone-side Output)**

You can set the sout gain using two methods:

- sediag: the label "mmfx_gsout_q10" maps to a read/write integer scalar which represents gain as a 16-bit integer [0..32767], where the gain g is represented as a fixed-point Q10 value:

$$g = \frac{q10}{2^{10}}$$

Thus q10 = 1024 represents unity gain (1024/1024 = 1), and q10 = 32767 represents 31.9 (= 32767/1024).

- C function: the functions
  `mmif.h::MMIfGetSoutGainQ10(int16 gain_q10)`
  `mmif.h::MMIfSetSoutGainQ10(int16 gain_q10)`
  get/set the gain of sout as a Q10 value.

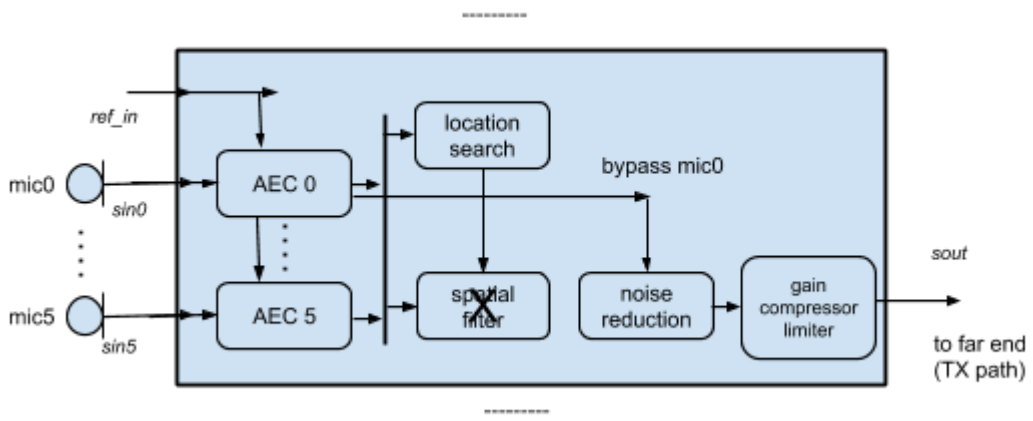
Monitoring Input/Output Power

Use the following sediags to monitor the signal power at various points in MMFx.

Name	Type	Length	RW	Description
sercv_rin_power_db_nn	float32	1	R	rin power: average block power in dB. Replace 'nn' with the channel number, e.g. sercv_rin_power_db_00 corresponds to rin channel 0.
sercv_rout_power_db_nn	float32	1	R	rout power: average block power per block in dB.
mmfx_sin_power_per_mic	float32	num_mics	R	sin power: average block power per microphone (linear measurement). This is an array of measurements, one per microphone.
aecmon_refin_power	float32	1	R	refin power: average block power (linear measurement)
aecmon_ave_erl_db	float32	1	R	ERL: the estimated ERL (echo return loss), averaged over all microphones. See "Testing Convergence" for an explanation of ERL.
aecmon_ave_erle_op_db	float32	1	R	ERLE: the estimated ERLE (echo return loss enhancement), averaged over all microphones. See "Testing Convergence" for an explanation of ERLE.
mmfx_sout_power_per_chan	float32	num_sout	R	sout power: average block power. This is an array of measurements, one per sout channel.

Bypassing the Spatial Filter (Single Mic Bypass)

Using sediag, you can configure MMFx to bypass the spatial filter; this effectively routes a single microphone through the spatial filter algorithm. Note that echo cancellation and noise reduction are still applied to the signal.



Example: Bypassing the Spatial Filter with Mic0

sediag Label	**Notes**
mmfx_bypass_spatial_filter	0 = no bypass (default) 1 = bypass spatial filtering
mmfx_bypass_spatial_filter_mic_index	0..(num_mics-1) The index of the desired mic

Enabling/Disabling Spectral Noise Reduction

The spectral noise reduction algorithm modifies the signal prior to output at sout. If necessary, you can entirely disable the noise reduction algorithm via sediag.

sediag Label	**Notes**
mmfx_senr_output_selector	SENR_APPLY_NOISE_REDUCTION: apply spectral noise reduction SENR_DISABLE_NOISE_REDUCTION: entirely bypass noise reduction, including residual echo suppression, noise floor suppression, and noise reference suppression

Enabling/Disabling Specific Microphones

A gain stage attached to each microphone allows you to selectively enable or disable single microphones via sediag.

Disable a microphone by setting the corresponding gain value to 0. Note that location search and spatial filtering require a specific set of gain values for your project to work correctly, so you will need to restore these gain values when re-enabling a microphone.

sediag Label	Type	Length	Notes
preproc_gsin_per_mic	float32 array	# mics	Example for 2 Mics: All mics enabled: [1, 1] All mics disabled: [0, 0]

Forcing a Specific Spatial Filter (Bypassing Location Search)

By default, the spatial filter forms a beam in the direction specified by the location search algorithm. However, you may want the spatial filter to "dwell" in a specific direction. The easiest way to force the spatial filter in a specific direction is to temporarily override the location search algorithm.

In order to override location search, you must set three sediag variables:

sediag Label	Type	Notes
mmfx_bypass_location_search	int16	0 = enable automatic location search (default) 1 = disable location search (use fixed direction)
mmfx_force_location_index	int16	When location search is bypassed, this variable specifies the desired spatial filter index. See spatialfilterconfig.c for solution indices.
mmfx_force_location_confidence	int16	An integer in the range [0..32767]. Typically, a lower confidence selects a wider spatial filter; a high confidence selects a narrow spatial filter.

The spatial filter solution indices are (typically) documented in the file project/projectname/spatialfilterconfig.c. For example:

```
/*
  | id  |  x (mm) |  y (mm) |  z (mm) |  description    |
  |-----|---------|---------|---------|-----------------|
  |   0 |   433.0 |     0.0 |   250.0 |  spatial beam 0 |
  |   1 |   375.0 |  -216.5 |   250.0 |  spatial beam 1 |
  |   2 |   216.5 |  -375.0 |   250.0 |  spatial beam 2 |
  |   3 |     0.0 |  -433.0 |   250.0 |  spatial beam 3 |
  |   4 |  -216.5 |  -375.0 |   250.0 |  spatial beam 4 |
  |   5 |  -375.0 |  -216.5 |   250.0 |  spatial beam 5 |
  |   6 |  -433.0 |    -0.0 |   250.0 |  spatial beam 6 |
  |   7 |  -375.0 |   216.5 |   250.0 |  spatial beam 7 |
  |   8 |  -216.5 |   375.0 |   250.0 |  spatial beam 8 |
```

```
  |   9 |    -0.0 |   433.0 |   250.0 |  spatial beam 9 |
  |  10 |   216.5 |   375.0 |   250.0 | spatial beam 10 |
  |  11 |   375.0 |   216.5 |   250.0 | spatial beam 11 |
  |  12 |  1000.0 |     0.0 |     0.0 |   Fallback Omni |
*/
```

To return to automatic location search/automatic beam selection, set mmfx_bypass_location_search = 0.

The same functionality is available through a C function; see mmif.h:

```
void MMIfSetManualLocation( uint16 locationIndex, int16 confidence )
void MMIfSetAutoLocationSearch( void )
```

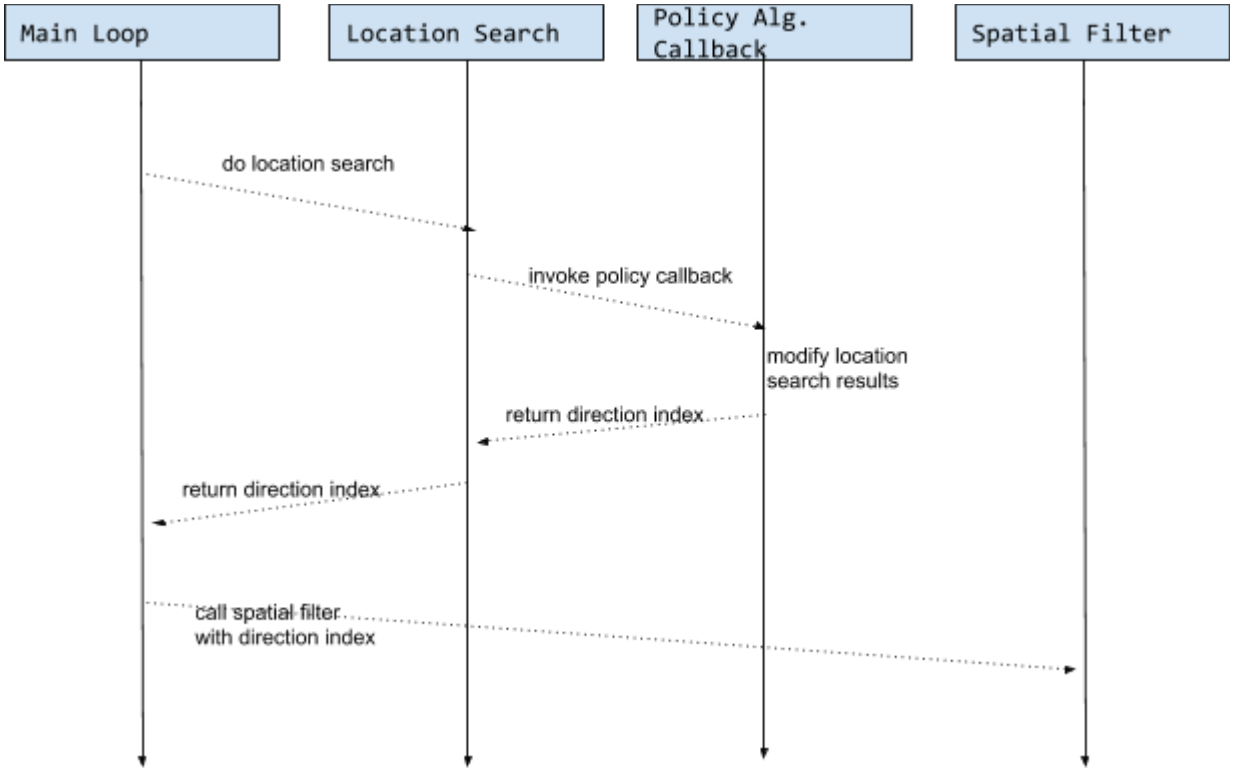
Policy Algorithm Integration

The "policy algorithm" refers to customer-specific code which modifies MMFx behavior in response to events. For example, consider a MMFx-enabled application which detects a keyphrase (or "wake-up word"). The keyphrase detection is implemented outside of MMFx. In response to detecting the keyphrase, the application may want to modify the location search and spatial filtering so that the spatial filter temporarily dwells in the direction of the keyphrase's arrival. This behavior modification is accomplished via the policy algorithm.

The policy algorithm is considered to be part of the MMIF layer, so you (the customer) own this code.

MMFx implements the policy algorithm with a callback: you implement your behavior in a function, located in policy_actions.c, and register the callback function with MMFx. MMFx will execute the callback function every sample block after the location search.

Within the callback function, you use sediag to to read and set parameters which control the behavior of MMFx. The diagram below illustrates how the policy algorithm callback could be used to modify the location search result, which in turn modifies the spatial filtering behavior.



Custom Policy Algorithms

Assuming that your project requires a policy algorithm, it is expected that customers will need to modify and tune the policy algorithm on their own. In order to do this, MMIF includes a method of overriding the default policy algorithm.

The policy algorithm source code is typically delivered in the files policy_actions.c and policy_actions.h. There are two functions:

- an initialization function, e.g. `void InitPolicyActions()`
- a callback function, e.g. `void DoPolicyActions(void *p)`

The init function is called during MMIfInit() and initializes any data structure used by the policy algorithm. It can accept any number of arguments (or no arguments).

The callback function must have the form:

```
// policy actions callback definition
void PolicyCallback(void *pArg);
```

In the MMIF layer, the policy callback function is registered during initialization with a special function:

```
void MMIfSetPolicyActions(MMIfPolicyFunc_t callbackFunc,
                          void *pArg);
```

This function also takes a void pointer, *pArg, which will be passed to the callback function when it is invoked. The pointer can point to a data structure for passing arguments or state information.

In order to override the default policy algorithm with your own, follow these steps:

1. Copy and rename `policy_actions.[ch]`, e.g. `my_policy.c` and `my_policy.h`
2. In `my_policy.[ch]`, rename the init and callback functions, e.g. `InitPolicyActions()` and `DoPolicyActions()`are renamed to `MyInitPolicyActions()` and `MyPolicyActions()`.
3. Modify the `MyInitPolicyActions` and `MyPolicyActions` functions as needed.
4. In your audio application, **after** calling `MMIfInit`(), initialize your policy code by calling `MyInitPolicyActions()`.
5. In your audio application, **after** calling `MMIfInit`(), register the new callback with `MMIfSetPolicyActions()` by supplying the callback function and an (optional) argument pointer, e.g.

```
MMIfInit();
MyInitPolicyActions();
MMIfSetPolicyActions(MyPolicyActions, &myPolicyState);
```

5. Compile `my_policy.c` to product the object file (`my_policy.o`).
6. Rebuild your audio application, supplying both the MMFx library and `my_policy.o`. Your application should now be using the custom policy callback.

Signal Recording

In order to debug what's going on with the system, Signal Essence will request raw recordings of **rin**, **rout**, **sin**, **sout** and **ref_in**. These recordings need to faithfully represent the signals that go to and come from the analog codecs and the remote side. We request a minimum record duration of at least 10 seconds on all channels simultaneously.

If ref_in is simply a delayed version of **rin**, it is not necessary to record **ref_in** separately (see Providing a Reference Signal to MMFx)

HARDWARE CONSIDERATIONS

Hardware Design

To achieve optimal performance with any audio system, and especially one that has an acoustic echo canceller (AEC), it's imperative that the entire signal chain from the MMFx through the DAC, amplifier, loudspeaker and acousto/mechanical system, and back in through the microphones, ADC and back to MMFx be carefully designed with attention to detail at every level, including electrical (EE) design, firmware implementation, acoustical design, industrial design, and mechanical design.

This article discusses only the basic EE design considerations and some firmware considerations for the ADC, DAC, and power amplifier sections. Signal Essence can assist you with all aspects of the hardware system design, from industrial design support through the entire design process to production testing support.

The general approach is to keep everything in the system as linear as possible, entirely avoid signal clipping at every stage, and put MMFx as close to the ADCs and DACs as possible, with minimal additional software layers in between.

Analog-facing signals `rout`, `sin`, and `ref_in` must be very tightly controlled with NO nonlinear processing in the DAC, power amp, or ADC channels. If there is going to be nonlinear processing like bass-enhancement or time/level varient equalization, it must be reflected in the `ref_in` signal. The signals `sout` and `rin` that come from the network side are non-critical, and can have other nonlinear processing (jitter buffers, G.xxx codecs, etc); perfection there is not critical for MMFx performance.

DAC and ADC (codec) Clocking

The clocking on the analog to digital converter (ADC) and digital to analog converter (DAC) is critical to MMFx performance. The ADC and DAC must run off the same clock source, and the sample rates must be integral multiples of each other. Generally, our systems run both the ADC and DAC at 16 kHz or 8 kHz. If 48kHz sampling is required (i.e. for music playback), this can be implemented on either ADC, DAC, or both. The 48kHz signal will be split and downsampled for the MMFx algorithm. When music is playing, 48kHz will be played out the DAC, and either 48kHz or 16kHz data will be read in from the ADC. In either case, the MMFx algorithm will still run at 16kHz.

It is not acceptable to let the ADC and DAC free run relative to each other, even if they have the same nominal sample rate. This often happens in USB systems where the DAC is on one USB crystal and the ADC is on a different crystal. If separate crystals will run the ADC and DAC systems, then a separate software module with phase-locked loops and asynchronous resampling will be required. That is outside the scope of this document and the MMFx algorithm.

ADC Phase/Delay Alignment

Having deterministic delays and phases between microphones are critical in microphone arrays; it's important that all microphone ADC channels are in perfect phase alignment with each other. This is generally true of the Left and Right channels of a stereo ADC, but it is not true of 2 identical ADC/CODEC chips when building a multi microphone array. If special consideration is not given in the hardware design of the ADC circuitry, it's possible to design a system that cannot have deterministic delay among all the ADC channels. Allowing ADCs to share the same MCLK, BCLK and WCLK does not guarantee phase alignment. You must check with the ADC vendor to ensure there is a method of phase aligning multiple chips. Some ADCs will require software controlled clock gating to achieve perfect alignment (eg, TI TLV320AIC3x), whereas others have a dedicated SYNC pin that runs between codecs to achieve alignment (eg Cirrus CS53L30).

In order to check and verify the alignment, it's best to send an electronic signal into the ADC inputs (thus bypassing the microphones). This can be a signal sent from just about any signal generator, just attenuated so as to not max out the ADC inputs. After recording all **sin** inputs, check the phase to ensure perfect time alignment (within 4 microseconds). The phase can be checked with an FFT, or with other audio analysis software. Please contact Signal Essence if assistance is needed.

Microphone & ADC Noise

Analog microphone signal levels are extremely low (as low as a few microvolts), and relatively high impedance (1 kOhm). Therefore they are subject to interference from any number of sources, including 50/60 Hz hum, RF, GSM noise, etc. It's important to use very good EE and PCB design techniques and components to minimize interference. Digital microphones can also have interference issues, and there are fewer ways to fix the digital signal if there is a problem. That said, digital microphones can be attractive in certain circumstances.

Deterministic Latency

The DAC for the loudspeaker must operate at the same sample rate as the ADC (or an integer multiple thereof), and should have a deterministic latency relative to the ADC signal. Some systems, including Windows and Linux, exhibit different latencies each time the audio system starts up. This is highly driver/system dependant, and must be addressed in your system. Note that the overall latency is not critical, only that it's deterministic. Lower latency is better for telephone call naturalness, but higher latency does not affect the MMFx performance until it gets to be very long.

Whether or not latency is deterministic when the system starts, it must remain constant while the system runs. Some platforms add or remove latency when running by dropping/adding samples. This is unacceptable for MMFx, and must be resolved prior to MMFx integration.

Power Amplifier (PA)

The gain of the DAC and power amplifier combination should be set such that driving a maximum value of +-32767 to the codec (assuming 16-bit words) will NOT drive the loudspeaker or power amplifier into clipping. This will need to be verified both at nominal supply voltage, and also at low supply voltage, such as low battery. This is mandatory if the ref_in signal to the AEC is managed in software (see below). Stated another way, the DAC, power amplifier, and loudspeaker system must be linear and time invariant if ref_in is derived in software.

BRING UP PROCEDURES

MMFx Integration Steps

Integration of MMFx into a target system requires integration both in hardware and software. Here is a checklist of steps that need to be accomplished for a successful integration:

1. Run a ‘ramp test’ (described below) to ensure that the ADC and DAC drivers are working properly.
2. Create a method to record **rin**, **rout**, **sin**, **sout**, and **ref_in** signals simultaneously.
3. Ensure that all ADC channels are time/phase aligned.
4. Verify deterministic ADC/DAC latency.
5. Set signal levels to/from loudspeakers & microphones properly. This includes checking for all analog gains in the entire ADC DAC pathway, including those inside and outside the codecs, like loudspeaker amplifiers.
6. Verify that any class-D amplifiers don’t have bad distortion caused by undersized ferrite beads. Size the ferrite beads to handle at least **2 * VCC_POWERAMP/DCR_SPEAKER** amps.
7. Verify that class-D cables do not run near or parallel to microphone cables.
8. Integrate MMFx into your application.
9. Support the MMFx Diagnostic interface (sediag), as described in this document.
10. Supply sample blocks to the MMIF functions every 10 ms.
11. Verify echo cancellation.

Each step builds upon the previous one; it’s critical the signals' hardware and driver paths are verified before you integrate and test with MMFx.

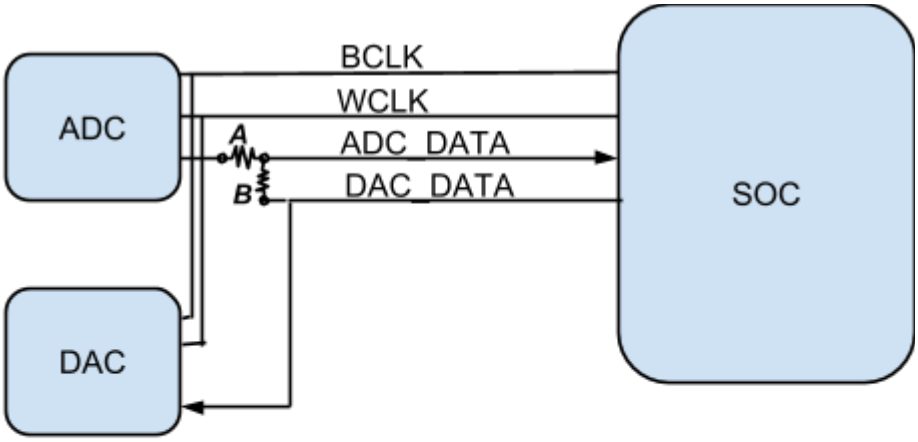
Ramp Test for Firmware/Driver Validation

We encounter systems where the audio drivers seem to work when playing or recording, but fail when running MMFx. This can occur when drivers/software miss samples or otherwise hiccup; these errors can go unnoticed when simply playing and recording audio. These types of problems can cause MMFx to fail unpredictably, even after hours of normal operation.

One of the most useful tests to run when developing audio drivers and hardware is the simple ‘ramp test.' Running this test or its equivalent before integrating MMFx into the system will save countless hours of debug time.

To perform the ramp test, send a digital ‘ramp’ or 16-bit incrementing counter (0, 1, 2, … 65535, 0, 1, 2…) out to the DAC. The hardware loops back the data-out pin to the data-in pin. The firmware then reads the digital data back and verifies that a delayed, bit-exact ramp comes back. The test should be run for an extended time, on the order of hours with varying system load. In order to facilitate this, the hardware should have a jumper setting to disconnect the

ADC data output and connect the DAC's data input channel to the CPU's data-in (see figure below). Alternatively, some systems have a built-in loopback for doing exactly this kind of test.



If the system does not have built-in loopback, you can add it easily at the hardware level by adding 2 zero ohm resistors ***A*** and ***B*** in the audio path that will disconnect the ADC and loop DAC data back to the ADC input.

For normal operation, install resistor ***A***. For loopback, remove resistor ***A*** and install resistor ***B***.

Ramp Test configuration

If the sample rates are asymmetric, for example if the ADC runs at 16kHz, but the DAC runs at 48kHz, then a simple looback will not be possible. In this case, you will need to verify each direction independently, using external verification hardware, such as an external codec.

Setting Signal Levels

Setting signal levels properly is critical and should be done early in the integration process. You should also maintain an easy way to test and verify signal levels for the duration of product development.

Equipment required:

- Ability to record loudspeaker output and analyze the distortion. This can be as simple as a (good quality) sound card and good quality omnidirectional microphone, combined with an FFT, or as sophisticated as an Audio Precision or Listen SoundCheck setup. Signal Essence can help you get this setup right.
- Sound level meter. Extreme precision is not required but +- 1 dB is expected; a smartphone-based sound meter should suffice, such as Audio Tools for the iPhone.
- A loudspeaker/power amplifier, separate from your target system, capable of playing a 1kHz tone. (A computer's amplified speakers should be sufficient)
- Ear plugs

Setting the Loudspeaker Level (rout)
It is absolutely critical that the loudspeaker and amplifier do not clip or cause any undue distortion. The AEC can not cancel nonlinear distortion (i.e. anything that falls into THD+N measurements).

The loudspeaker pathway gains should be set to maximize the dynamic range of the loudspeaker and power amplifier without sending the amplifier into clipping. Often the power output of the power amplifier is specified at 10% THD. This is already driving the amplifier into clipping and is definitely above what would be good for the AEC. You should use the spec at 1% for the hardware design and testing.

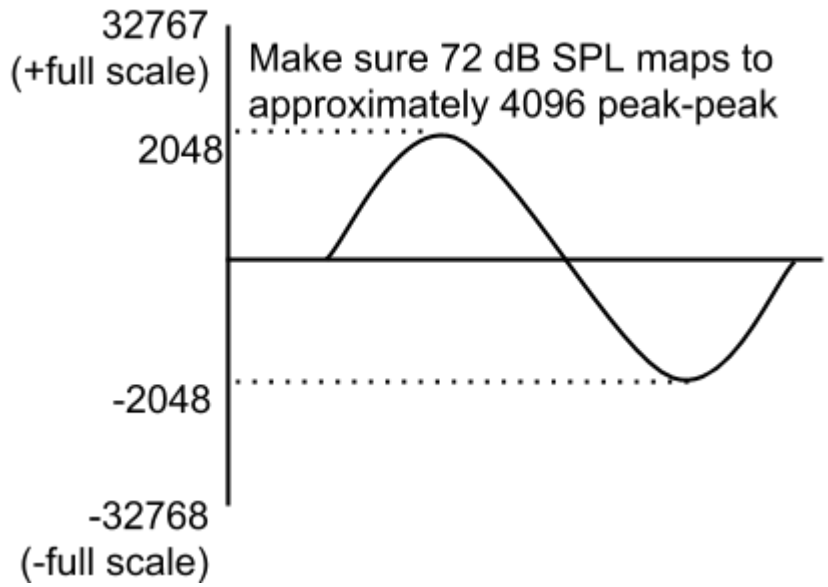
If you have fine grained control over the DAC/PA gain, set it so that when you output a +- 32767 sine wave to the DAC, the loudspeaker does not clip. If it is impossible to set the gain of the DAC path to such a fine-grain, you can set it so that the hardware can clip, but then we can limit the output level in the MMFx algorithm. This will reduce the dynamic range of the system somewhat, and can cause problems if the limiter is not set right, but should not affect system performance too badly once everything is tuned properly.

Signal Essence can provide a standalone tone generator software module to make this test easy.

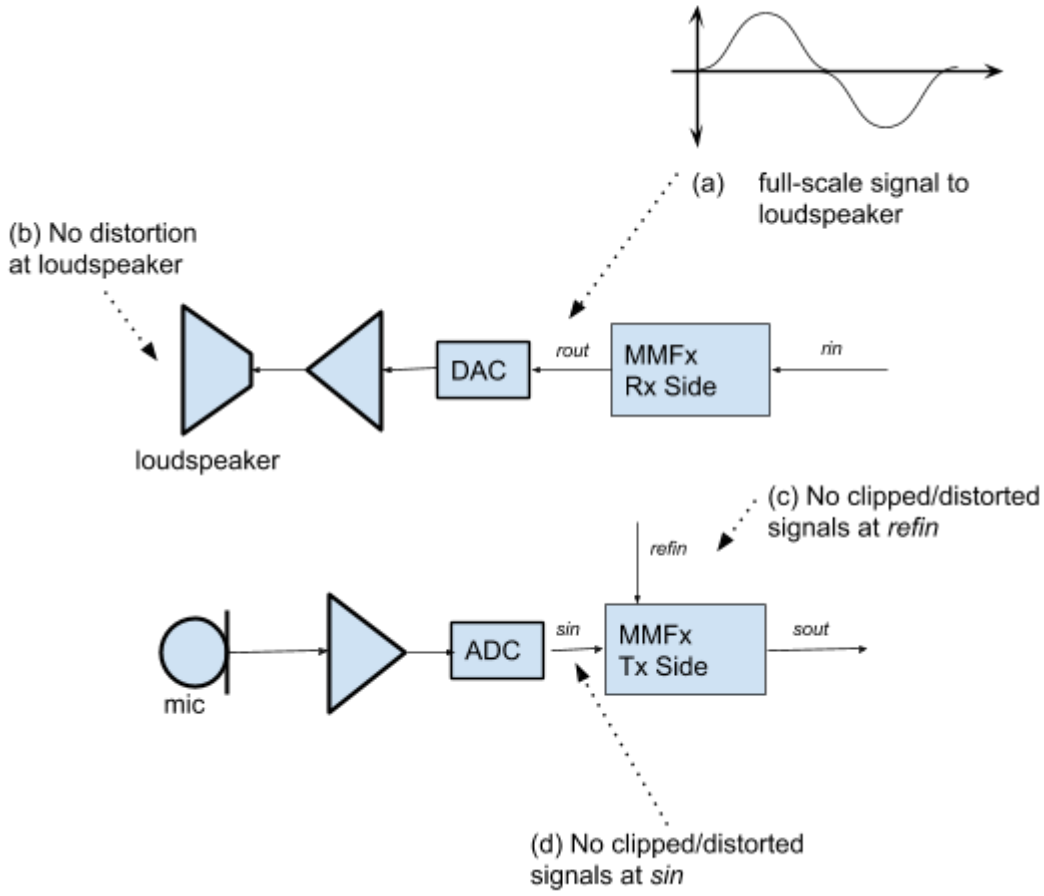
Once you believe that all gains are set properly, put your ear plugs in and play a full-scale sine wave out to the loudspeaker and record/analyze the sound for distortion. Ensure that distortion is under the loudspeaker's specification at its rated power input.

Setting the Microphone Level (sin)
For systems with analog ADC gain, the microphone level must also be set appropriately for the MMFx algorithm to work.

Set up the test loudspeaker about 50 cm away from the target system's microphone. If it is a directional microphone, make sure the loudspeaker is in front of the microphone. Put the sound level meter next to the target system's microphone. Play a 1 kHz tone and adjust the volume until the sound meter reads 72 dB SPL. (At 1 kHz the sound meter will read the same regardless of A or C weighting setting). Record the **sin** signals and verify that the signal level is approximately 4096 peak to peak. If not, adjust the gain the re-record the signal to verify the proper levels.



Sin Signal Level



Testing Convergence

Once MMFx is integrated into your system and audio is flowing through the loudspeaker (rout), reference (ref_in) and microphone (sin) paths, you can try exercising the Acoustic Echo Cancellers (AECs) to determine if the cancellers properly converge and cancel echo.

MMFx includes a signal generator which will inject a white noise into rout (as well as sout, though this does not affect AEC operation). The signal generator is controlled via two functions declared in mmif.h:

```
void MMIfEnableWhiteNoiseInjection(void);   // enable white noise injection
void MMIfEnableNormalMode(void);            // disable noise injection
```

In your code, call MMIfEnableWhiteNoiseInjection() to enable white noise injection. You should hear white noise coming out of the loudspeaker.

During white noise injection, use the sediag interface to monitor two entries:

1. "aecmon_erle_deep_db_per_mic": this entry represents a float32 array, one element per microphone, containing the ERLE (Echo Return Loss Enhancement) in dB for each microphone. When working correctly, the ERLE should drop from its initial configured value to -15 .. -30 dB within 5-10 seconds after the white noise starts.
2. "aecmon_ave_erle_deep_db": this entry represents a single float32 scalar, representing the average ERLE across all microphones.

Call `MMIfEnableNormalMode`() to disable white noise injection.

If the AECs fail to exhibit convergence, make and send audio recordings to Signal Essence. The recordings should include, at a minimum, ref_in and sin (all microphones).