# *McASP Design Guide - Tips, Tricks, and Practical Examples*

*Bobby Tufino*

**ABSTRACT**

McASP is TI’s Multichannel Audio Serial Port. It was designed specifically to support use cases requiring multichannel, multi-zone audio, and has been designed into many audio products, such as Audio/Video Receivers, sound bars, automotive amplifiers, and infotainment systems. This application report is intended for engineers who are new to designing audio systems featuring McASP.

**Contents**

**List of Figures**

## Trademarks

## 1 Introduction

One of the most useful things about McASP is its flexible clocking scheme, which allows for complete independence between the receive and transmit ports, but that flexibility means that the engineers implementing an audio system with McASP will have to make some important design choices. These are not always obvious and are best illustrated with some proven practical examples from audio products in the field.

The primary goal of this document is to make it easier for an engineer to determine how to connect a McASP (or multiple McASPs) to audio devices in their system. Although this is primarily a hardware design-focused application report, a few common pitfalls that engineers run into when programming McASP are also discussed.

First, a few disclaimers:

- This application report focuses on the hardware aspect of McASP, rather than the software aspect. Writing software for McASP comes with its own challenges, but before an engineer gets to that point, it has to be wired up properly, which is our focus.
- McASP is typically used to interface with devices using a time division-multiplexed (TDM) protocol, and in most cases those devices will be using a specific configuration of TDM called Inter-IC Sound (I2S). It is assumed that I2S is being used in all examples provided in this document, but that is somewhat arbitrary. The physical hookup of McASP to devices using multi-slot TDM or using I2S is the same.
- This document is not meant to be a detailed specification. Further, this document is intended to cover McASP in general, rather than McASP on a specific TI device. It covers the key characteristics of McASP at a high level, but the device-specific Technical Reference Manual (TRM) is still the best place to go for architectural and chip-level details.

## 2 McASP - External Signals

### 2.1 Data Pins

McASP may have up to 16 serializers. These serializers are connected to data pins, referred to as AXR pins. They are named as such ("audio transmit/receive" = "AXR") because any McASP data pin can be configured to functions as an input or as an output. Note that if an AXR pin is running as a transmitter, it is necessary to re-initialize the McASP in order to re-configure it as a receiver. Dynamically switching direction during operation is not supported.

Some devices do not pin out all 16 possible data pins on all McASPs. For instance, a device may have one McASP with 16 data pins, one with 10, and one with six. Further, some devices may have one McASP’s data pins multiplexed with another McASP's data pins. These details are captured in the device-specific data sheet and TRM.

Often times, a TI device has multiple instances of McASP, and the number of data pins required may determine which McASP a designer chooses to connect to a particular audio device in the system. Consider a scenario where the system requires 12 channels of I2S audio. Since each data pin will carry two channels’ worth of audio, this means that six data pins are required. The designer must choose a McASP that has at least that many data pins available.

The McASP data pin hookup is straightforward. Since any McASP AXR pin can be configured for either transmit or receive, the designer is free to choose whichever pins are most convenient for the system.

### 2.2 Mute Pins

McASP can have up to two mute pins:

- AMUTEIN – This is an input. Some external devices have a mute output pin; such a signal can be connected to AMUTEIN. McASP can be configured to mute I2S output under this condition.
- AMUTE – This is an output that McASP can be configured to drive under specific error conditions. For more details, see the device-specific TRM.

Mute pin hookup is also straightforward. Configuration of the AMUTE pin’s behavior takes some planning (see your device-specific TRM), but it is easy to figure out where to connect it. If the downstream device has a mute input pin, that is what AMUTE should be connected to.

## 2.3 Clock Pins

McASP may have up to six clock pins, all (and any) of which can be generated internally or sourced externally. There are three types of clock signals on McASP: high frequency (AHCLK), bit (ACLK), and frame sync (AFS). However, each has a transmit version (X, for example, AHCLKX) and a receive version (R, for example, AFSR).

Every McASP has a receive clocking section and a transmit clocking section. These are sometimes referred to as the receive port and the transmit port, and each clock port constitutes a clock zone. Therefore, each McASP has two potential clock zones. These can be run asynchronously with respect to each other, but in some cases, synchronous operation is appropriate.

- AHCLKX and AHCLKR – These are high-frequency clocks pins, sometimes referred to as master clocks (often referred to on audio codecs as MCLK). McASP uses a master clock for one purpose: to divide it down and generate a bit clock. There are several cases where a master clock is not required.
- ACLKX and ACLKR – These are bit clocks, often referred to on audio devices as BCK. Data is clocked in and out with respect to bit clock edges. Furthermore, much of McASP's internal logic (state machines, and so forth) runs off of the bit clock, so a bit clock is ALWAYS required.
- AFSX and AFSR – These are the frame sync clocks, often referred to as word clocks, or more commonly as left-right clocks (LRCK). The "left-right" terminology comes from stereo audio in the I2S format, in which the edges of the frame sync clock denote the bits corresponding to the left and right channels. The frame sync clocks run at the audio stream's sample rate.

Figure 1 is a simplified representation of McASP's external signals, omitting the mute pins.

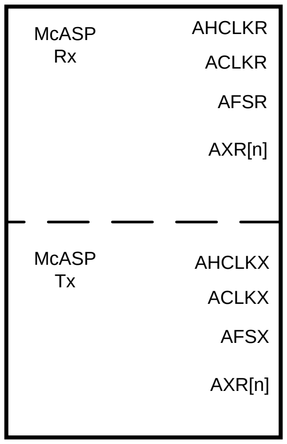

**Figure 1. External McASP Signals**

## 3 Clock Generation With McASP

In some cases, McASP is required to act as a clock master and must generate clocks. Figure 2 is a simplified block diagram of McASP's clock generation architecture. For more details, see the device-specific TRM.

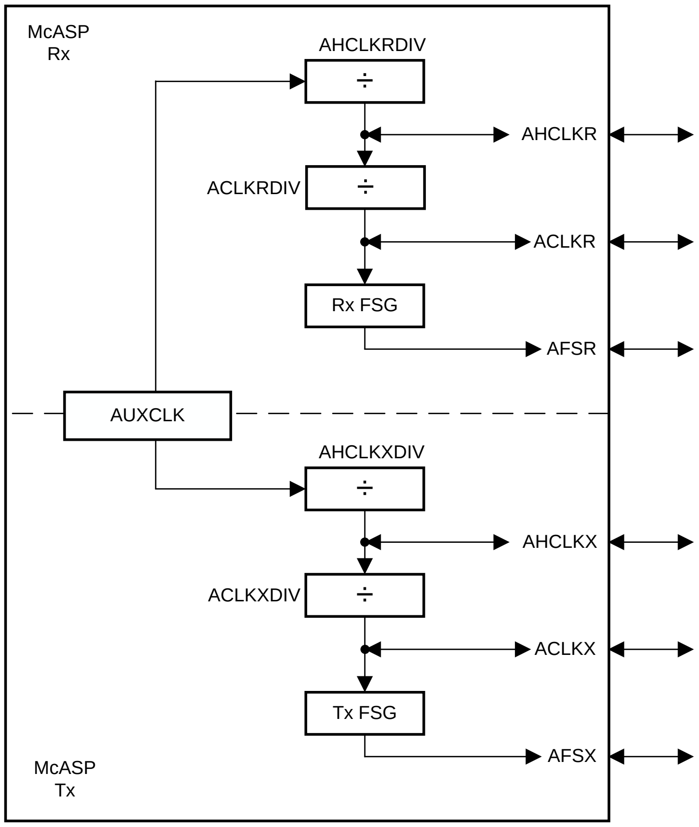

**Figure 2. McASP Clock Generation Architecture**

The block marked AUXCLK in Figure 2 is a clock source that is available to both the Rx and Tx sections of McASP. The source of AUXCLK varies, depending on the device, but it is often derived directly from the device's main system clock source (usually an on-board oscillator or an externally generated square wave).

For both the receive and transmit sections, the AUXCLK is fed into an integer high-frequency clock divider: AHCLKRDIV or AHCLKXDIV. This clock signal can be divided down to generate the McASP's Rx or Tx master clock: AHCLKR or AHCLKX. This signal can be driven out on its corresponding pin, if configured to do so. It also feeds the next clock divider: ACLKRDIV or ACLKXDIV.

Similarly, the integer bit clock divider ACLK[R/X]DIV generates the McASP's bit clock: ACLKR or ACLKX. That signal can be driven out on the corresponding pin. Note that the bit clock divider can instead be fed by the AHCLK[R/X] pin. It is common, particularly when McASP is being used as a transmitter, to receive a master clock from an external source and divide it down to generate a bit clock, which is then used to generate a frame sync. An example of this is provided in Section 4.2.1.1.

The next stage of clock generation looks similar. These are the receive and transmit Frame Sync Generators (FSGs). The output of the FSGs will be AFSR and AFSX. These can also be driven out of their corresponding clock pins. The frame sync generator can be fed by an external bit clock provided to the ACLK[R/X] pin. This is not common, as the bit clock and frame sync are usually both internally generated, or both externally sourced.

A key point about the McASP's clock generation architecture is that its clock dividers are all integer dividers. This becomes very important when trying to figure out if it is possible to generate a specific clock frequency on-chip. This topic will be revisited in Section 5.2. For now, the point is that McASP has integer dividers (and frame sync generators) that can be used to generate internal clocks, and those internal clocks may be driven out of their corresponding device pins.

## 4 McASP Hookup - Practical Examples

McASP typically connects to audio devices that have the same types of clock pins (master, bit, and frame sync) mentioned in Section 3, so it is fairly easy to figure out which kind of McASP clock pins to a connect to another audio device's clock pins: bit clocks go to bit clocks, frame syncs go to frame syncs, and so forth. Most questions regarding McASP are about which clock port to use (receive or transmit) and which direction (input or output) the pins should be configured for. The easiest way to clear up this aspect of audio designs with McASP is to use some real-world examples.

An engineer that is new to digital audio system design might have to think twice about whether to hook their audio device to a McASP's receive or transmit port. The first thing to consider is which direction the data is going. If the McASP will receive data from the audio device, then that device's clock pins should be connected to the McASP's receive clock pins. It is that simple.

### *4.1 McASP as a Receiver*

#### 4.1.1 Analog-to-Digital Converter (ADC)

It is very common for an audio system, such as a soundbar or a car stereo, to have an analog input. In all such cases, an audio ADC is required. Once the A-to-D conversion has occurred, the resulting digital data stream must be fed to a McASP. Since the McASP receives data from the ADC, the McASP's Rx clock pins are connected to the ADC's clock pins.

At this point, the designer must consult the device-specific data manual for the ADC that they have chosen to use. Some ADCs offer the designer the option to configure them as clock masters or clock slaves, but some do not have that flexibility and must be configured as one or the other.

##### *4.1.1.1 ADC as Clock Master*

If the ADC must be configured as a clock master, then it will be the device generating all clocks. It has been established that the device will be connected to McASP's Rx clock pins, as shown in Figure 3.

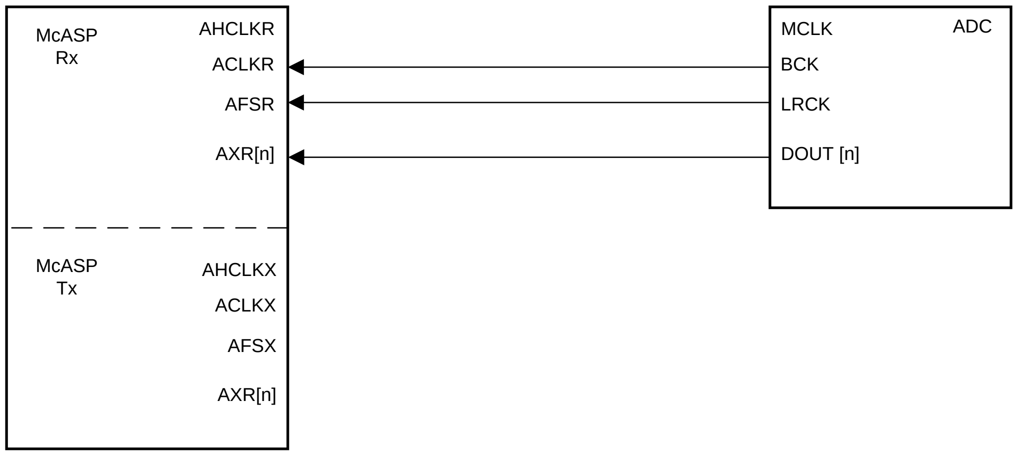

**Figure 3. ADC as Clock Master**

Important things to note:

- McASP's receive clock pins, ACLKR and AFSR, are operating as inputs. The ADC, as the clock master, is driving those clock lines. McASP must be configured as the clock slave, which means that the ACLKR and AFSR pins must be configured as inputs, and must be configured for an EXTERNAL clock source. These two things sound equivalent, but they are not. McASP pin direction and clock source are two different settings. For more information on how to configure this, see the device-specific TRM.
- AHCLKR is not used, only bit clock and frame sync are required. AHCLKR is only necessary if the McASP's Rx section is fed a high-frequency clock and must then divide it down in order to generate bit clock and frame sync. In the above example, this is not necessary, since there is no need for McASP to generate Rx clocks.

#### *4.1.1.2 ADC as Clock Slave*

Many ADCs do not have the ability to generate clocks themselves and must be operated as clock slaves. Since McASP receives data from the ADC, it is appropriate to configure its Rx clock pins as outputs, and generate clocks that will be driven into the ADC's clock pins.

Recall that McASP has integer clock dividers. If the device's AUXCLK runs at a frequency that can be used to generate all of the necessary McASP Rx clocks at the desired frequencies, then AHCLKR, ACLKR, and AFSR will all be generated internally and all configured as outputs, as shown in Figure 4.

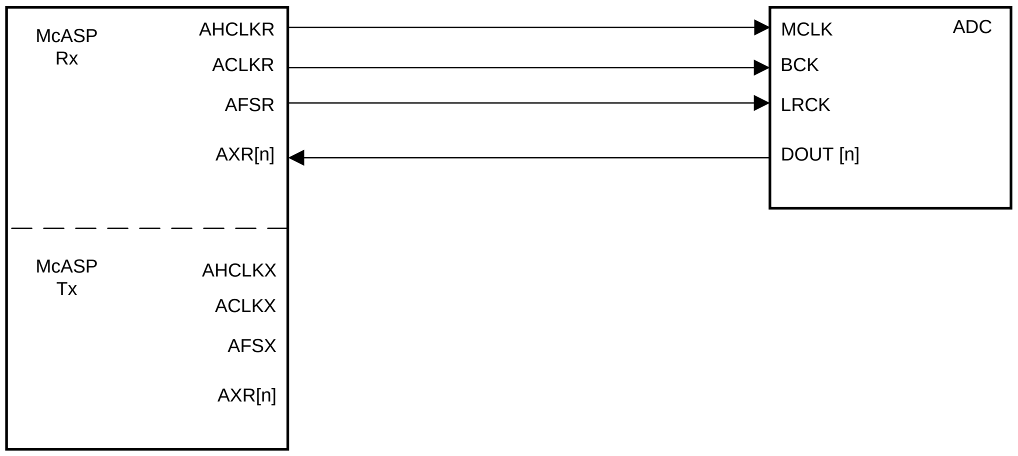

**Figure 4. ADC as Clock Slave, all McASP Clocks Generated Internally**

Important things to note:

- All McASP Rx clocks are outputs.
- AHCLKR is driven out of McASP. This is because ADCs usually require a high-frequency clock for operation. For more information, see the device-specific ADC data manual.

In this case, the clock generation scheme is as shown in Figure 5:

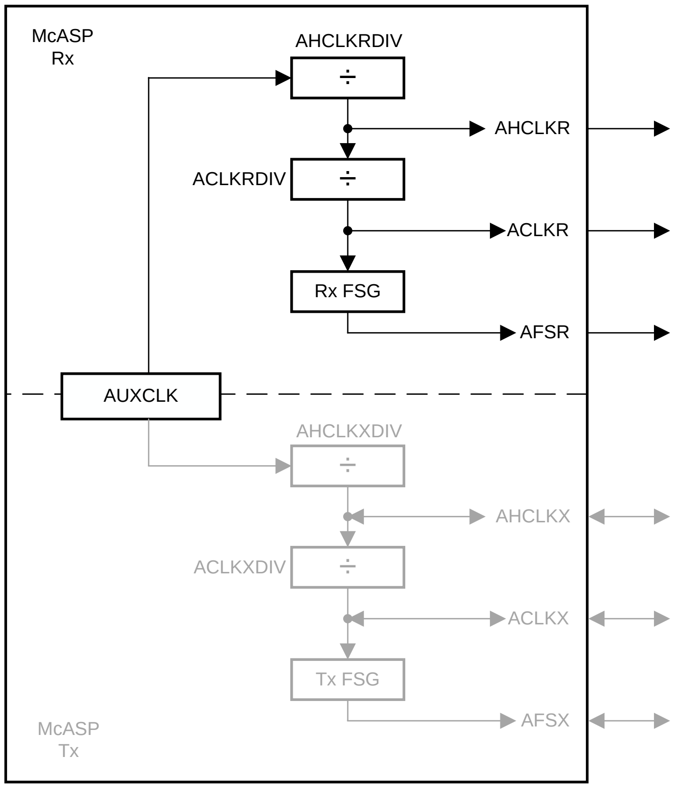

**Figure 5. ADC as Clock Slave, Rx Clocks Generated Internally**

Note that the AUXCLK feeds the AHCLKRDIV, which feeds ACLKRDIV, which feeds the Rx FSG.

In many cases, the AUXCLK is running at a frequency that cannot be divided down to the desired bit clock and frame sync rates (recall that McASP only has integer dividers - sometimes the math does not work out). In that case, an appropriate external clock source of some kind must be fed to the AHCLKR pin and then divided down to generate bit clock and frame sync. Figure 6 illustrates this scenario.

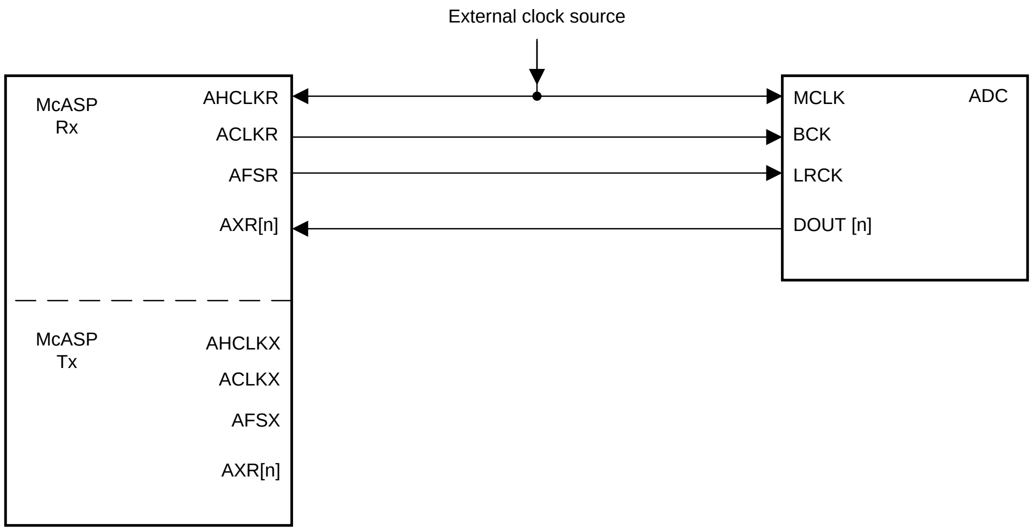

**Figure 6. ADC as Clock Slave, External Master Clock**

Note that the external clock feeds both the ADC's MCLK and the AHCLKR pin. Clock generation for this scenario is illustrated in Figure 7.

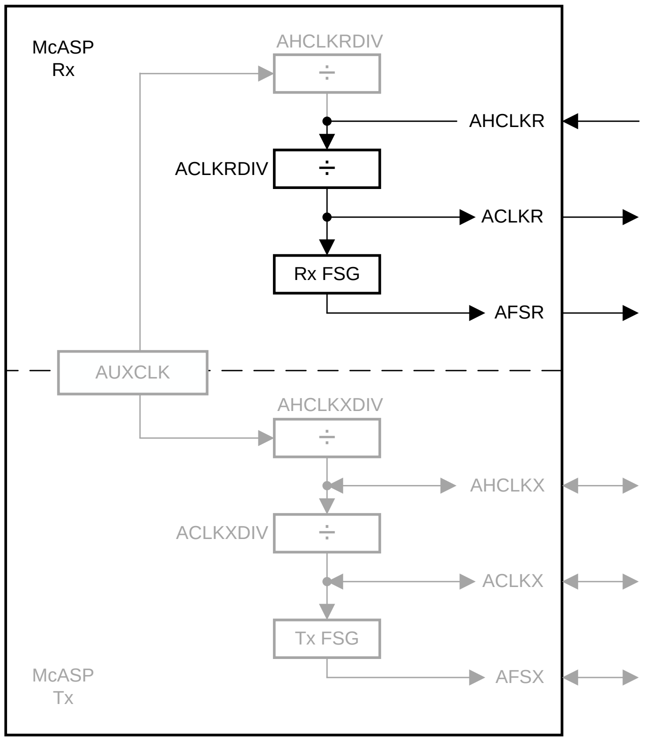

**Figure 7. ADC as Clock Slave, External Master Clock Used for Rx Clock Generation**

Note that in this case, AUXCLK is not used; therefore, AHCLKRDIV is not used. The external master clock source is used to generate the ACLKR and AFSR signals, which are then fed to the ADC's BCK and LRCK pins, satisfying the ADC's requirement to act as a clock slave.

## 4.2 *McASP as a Transmitter*

As discussed earlier, since McASP will be transmitting data, the Tx clock pins (AHCLKX, ACLKX, AFSX) will be used.

### 4.2.1 Digital-to-Analog Converter (DAC)

Any digital audio system that uses analog amplifiers will require DACs to turn the digital audio data into an analog signal, which will then be amplified. In most cases, the DAC expects to function as a clock slave. Many modern use cases, such as soundbars, avoid DACs completely and feed the digital audio data to a PWM modulator, which then feeds a digital amplifier. In that case, the hookup is nearly identical to what is shown in the DAC example. Even the pin names on the modulator will very likely be the same as what would be found on a DAC, as it also requires a master clock, bit clock, and frame sync.

The receiving aspect of McASP was discussed in isolation in Section 4.1, but this is not practical when discussing McASP as a transmitter. In almost all cases, McASP will be transmitting data that has been received from an external audio device, processed on the TI device's CPU, and then sent out to the DAC. In order to avoid the need for asynchronous sample rate conversion (ASRC), it is necessary that the McASP's transmit clocks be synchronous with respect to the receive clocks. For this reason, McASP's transmit clocks are very often derived from whichever master clock is used by the device that McASP receives data from.

#### 4.2.1.1 HDMI Transceiver + DAC

A practical example of this scenario can be found in any Audio/Video Receiver (AVR) on the market, where the device providing data to the McASP will be an HDMI transceiver. HDMI transceivers are clock masters with respect to McASP. The clocks for the DAC are derived from the HDMI device's master clock, as shown in Figure 8.

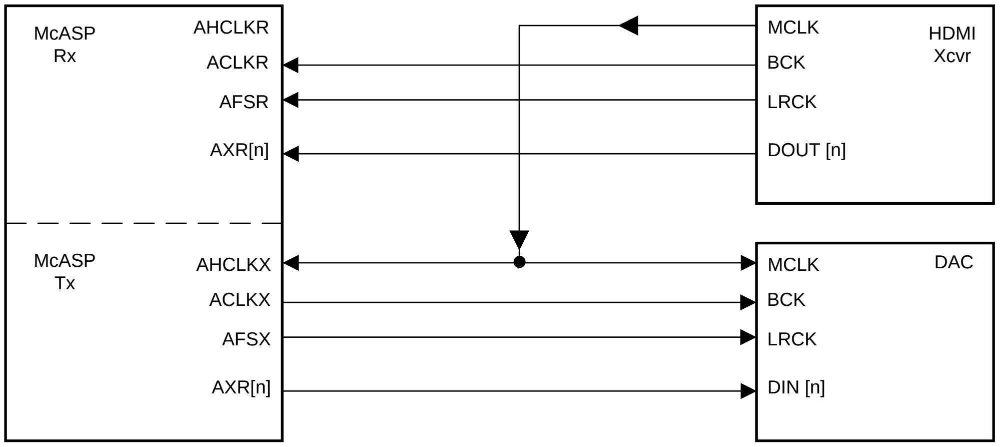

**Figure 8. McASP Receiver as Slave to HDMI Transceiver, McASP Transmitter as "hybrid master" to DAC**

Of note in Figure 8:

- The Rx section of the McASP is slave to the HDMI transceiver - AHCLKR is not used.
- The HDMI transceiver's master clock is fed to the AHCLKX pin and to the DAC.
- The Tx section of the McASP acts as clock master to the DAC, even though it only generates the bit clock and frame sync (the HDMI transceiver's MCLK is fed directly to the DAC). The details of this clock generation are shown in Figure 9.

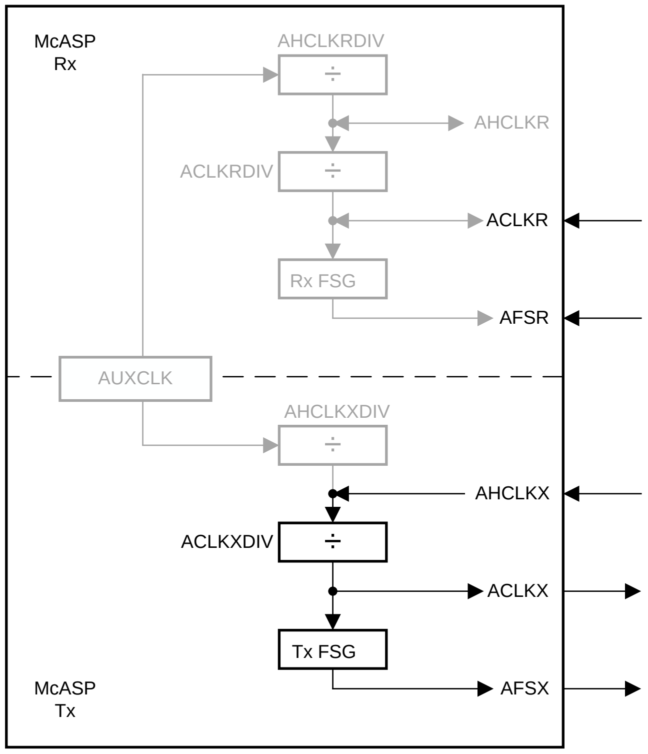

**Figure 9. McASP Receiver as Slave to HDMI Transceiver, HDMI Transceiver's MCLK Used for McASP Tx Clock Generation**

Note that AHCLKX receives an external master clock from the HDMI transceiver and uses it to generate ACLKX and AFSX, which are then fed to the DAC's BCK and LRCK pins.

The example in Figure 9 uses one McASP to both transmit and receive, but using multiple McASPs, as in the example in Figure 10, is quite common.

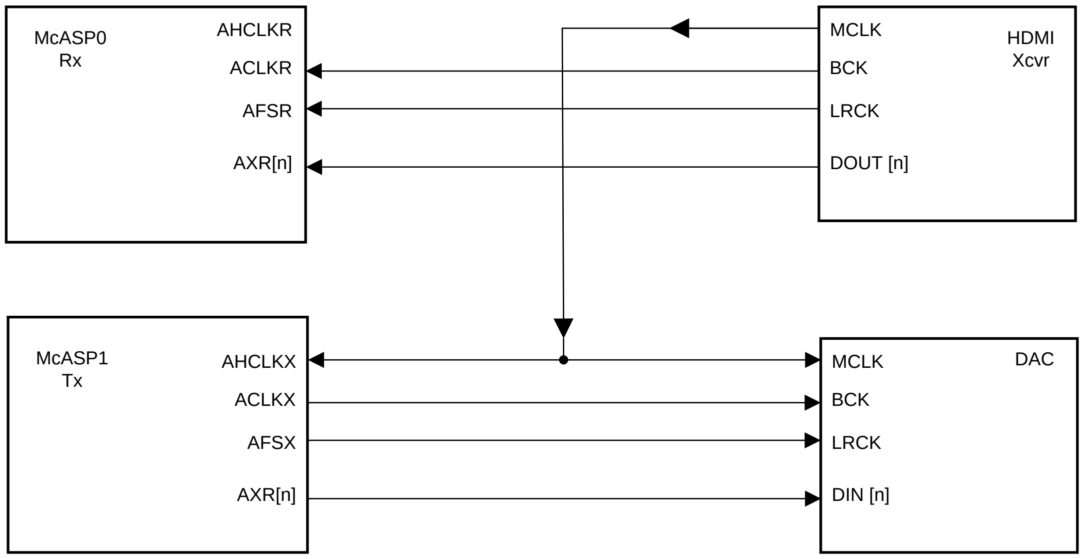

**Figure 10. HDMI Transceiver and DAC, Using Multiple McASPs**

The example used in Figure 10 is very similar to the example used in Figure 9, the only difference being that multiple McASPs are used. It is important to remember that even when dealing with different McASPs, it is still necessary for the transmitting clock section to be synchronous with respect to the receiving clock section.

Why use multiple McASPs? In the example used in Figure 10, perhaps McASP0's Tx pins were unavailable due to the device's pin multiplexing. More likely, there are multiple audio sources. McASP1 Rx could be used to receive data from an ADC, for instance, as in the example shown in Figure 11. Note that in a case such as that, a clock multiplexer must be used such that the transmitting McASP is always synchronous with respect to whichever source is selected.

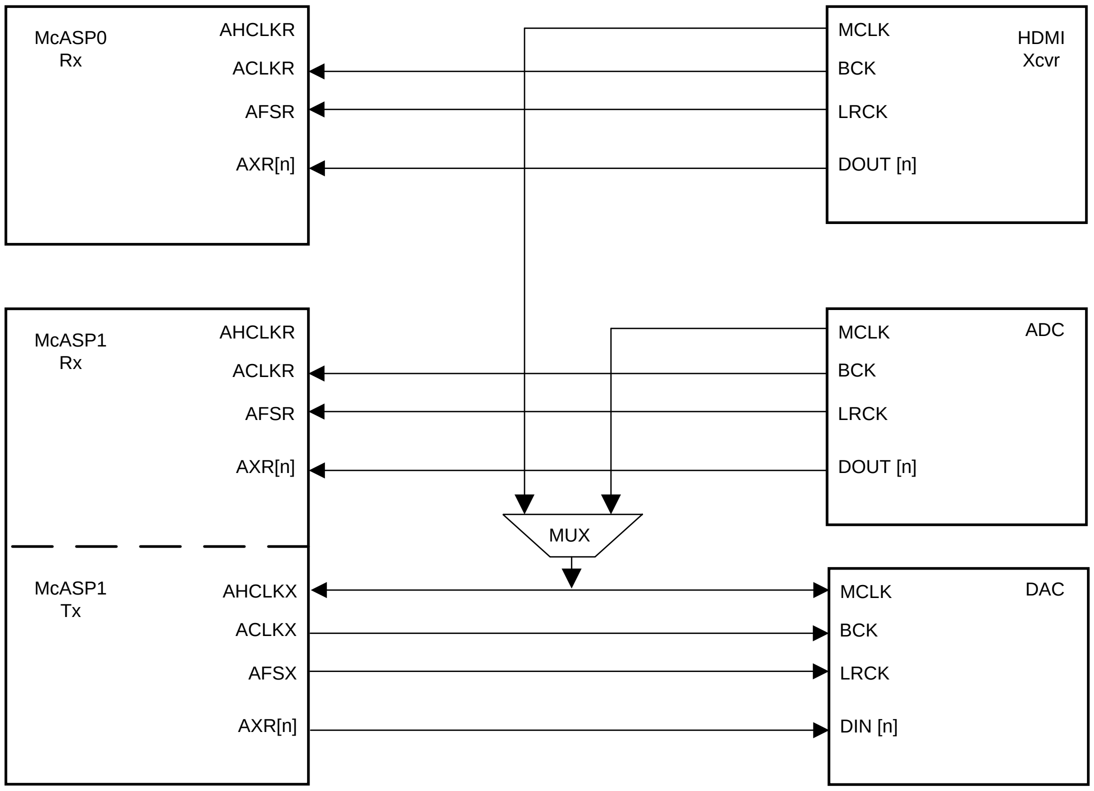

**Figure 11. HDMI Transceiver, ADC, and DAC, Using Clock MUX and Multiple McASPs**

In the example used in Figure 11, when McASP is transmitting audio that is derived from the ADC, the ADC's MCLK must be routed, via the clock multiplexer, to McASP1's AHCLKX. When the HDMI device is the source, its MCLK must be routed to McASP1's AHCLKX. In this manner, the synchronous relationship between receiver and transmitter is always maintained.

## 4.3 Audio Codec - SYNC Mode

There is one clocking use case that has not been covered thus far: SYNC mode. Although a McASP's transmit and receive clocking sections can be operated asynchronously, occasionally it is preferable for them to operate fully synchronously with respect to each other. A typical example is the connection to an audio codec with integrated ADC and DAC, as these very often do not have independent Rx and Tx clock sections. Very often, they will have data input pins and data output pins, but only one MCLK, one BCK, and one LRCK for the whole device. McASP's SYNC mode (accessible from the ACLKXCTL register) is very efficient in this case. For more information, see the device-specific TRM. The physical connection is illustrated in Figure 12.

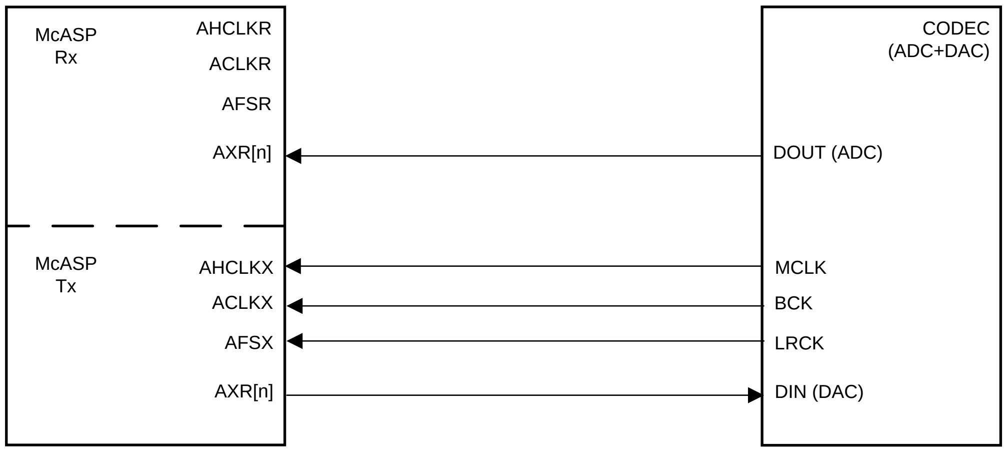

**Figure 12. Codec Acting as Clock Master to McASP (SYNC mode)**

Note that in Figure 12, the codec's clocks (it is the clock master in this example) go only to the transmit section of the McASP. Nothing is connected to the McASP receive section's clock pins. The connection of MCLK to AHCLKX is not strictly required, but connecting them does not hurt. There may be scenarios in which you want to switch the codec to slave mode (in case the transmitting McASP wants to be in control of the bit clock and frame syncs for upsampling) and having that MCLK signal available is useful.

It is not necessary to connect anything to the McASP's Rx clock pins because, in SYNC mode, the McASP's Tx clocks are internally routed to the Rx section, as shown in Figure 13.

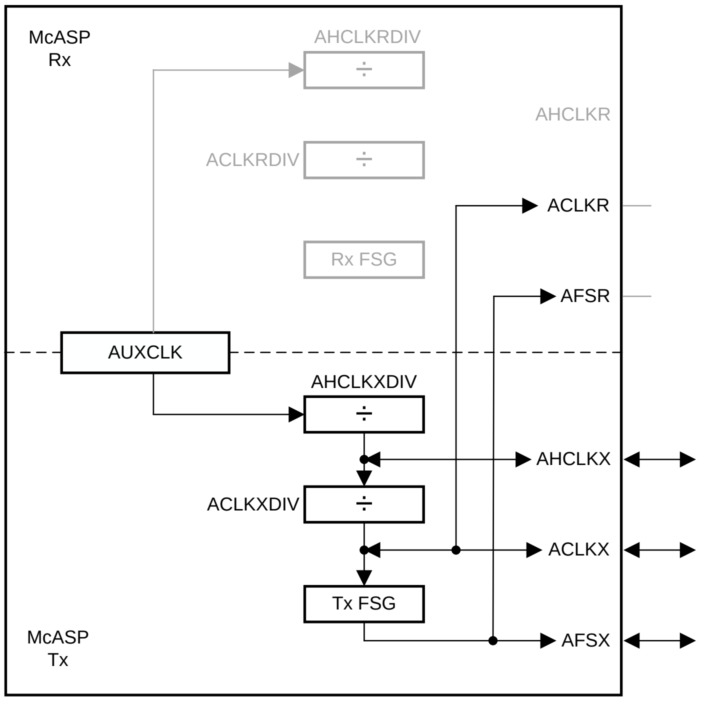

**Figure 13. Clock Generation in SYNC Mode**

Note that whether the Tx section's clocks are generated internally or externally, the bit clock and frame sync are always routed to the receive section for internal use only. When SYNC mode is used, those signals cannot be routed to the McASP's ACLKR and AFSR pins. They are only used internally to clock data in.

### *4.4 McASP Examples - Key Points*

The above examples all illustrate the same basic points:

- If McASP is receiving data, use the Rx clock pins. If it is transmitting data, use the Tx clock pins.
- Look at the external audio device's data manual to determine if it needs to be a clock master or a clock slave, then configure the McASP's clock pins accordingly. If the audio device can be configured as either, choose whichever configuration is easier (and more cost-effective) to implement.
- If McASP will be transmitting data that is derived from some external audio source, the transmitting McASP's clock section must be synchronous with respect to the receiving McASP's clock section.

## 5 Additional Topics

Although the focus of this application report is the physical hookup of McASP to other audio devices, there are a few software configuration topics that are asked about often enough to warrant some limited discussion.

### 5.1 *How Do I Configure a McASP for I2S Mode?*

Upon looking through the *McASP* chapter of the device-specific TRM, you may notice that there is not a register bit marked “I2S mode.”

As mentioned previously, I2S is a specific case of TDM Format (your device-specific TRM should contain a section named “Inter-IC Sound (I2S)" or something similar, with helpful graphics and descriptions). I2S has some key characteristics that must be selected in the McASP’s configuration registers:

- One audio frame consists of two slots (set in the AFSXCTL.XMOD or AFSRCTL.RMOD bitfield)
- Start of frame is denoted by the FALLING edge of the frame sync (set in the AFSXCTL.FSXP or AFSRCTL.FSRP bitfield)
- There is a one-bit data delay with respect to the frame sync (set in the XFMT.XDATDLY or RFMT.RDATDLY bitfield)

The full configuration process of a McASP is covered in the device-specific TRM. The above are simply some key points to keep in mind during configuration.

### 5.2 *How Do I Generate a 48 kHz Frame Sync With McASP?*

Different versions of this question are quite common, although sometimes the requested frequency is 96 kHz, or 44.1 kHz, or some other common audio frequency. In the below exercise, recall that McASP only has integer clock dividers.

It is easiest to work backwards, starting with the frame sync:

AFSX/AFSR = 48 kHz. What must ACLKX/R’s frequency be?

In the case of I2S, there are two slots per frame. Each slot consists of 32 bits. Even if the data payload is only 16 bits or 24 bits, it is encapsulated within a 32-bit word. Based on this logic:

2 slots * 32 bits per slot * 48 kHz = 3.072 MHz bit clock

The results so far:

- AFSX/AFSR = 48 kHz
- ACLKX/ACLKR = 3.072 MHz
- AHCLKX/AHCLKR = ???

The master clock is typically required to run at a multiple of the sample rate, fs: 128fs, 256fs, or 512fs, (the device-specific data manual for your audio device will tell you which one is expected). Therefore, if fs = 48 kHz, the master clock must run at:

128 * fs = 6.144 MHz, or

256 * fs = 12.288 MHz, or

512 * fs = 24.576 MHz

If the frame sync and bit clock are to be generated completely internally, without the use of an external clock source, then the AUXCLK (recall that node in the clock generation diagrams) must be running at one of the above mentioned three frequencies.

Unfortunately, it is somewhat rare for this to be the case, as TI devices usually do not use those fundamental frequencies to drive the main system PLL. Some older devices in the C6000 series do allow for 24.576 MHz to be used, and in that case, it is possible to generate 48/96/192 kHz internally, without the use of any external audio clock source. If not, then it is necessary to provide an appropriate master clock to the AHCLKX or AHCLKR pin and use the McASP’s integer dividers to generate the bit clock and frame sync. Sometimes, this requires adding a clock generator chip to the system and, sometimes, that clock is already available (generated by some other device).

In the end, you may or may not be able to internally generate the frame sync that you want. It depends on what AUXCLK frequency is available on your TI device.

## *5.3 How Do I Properly Set Up the AFIFO?*

On many TI devices, McASP has an audio FIFO buffer (AFIFO). It provides additional buffering, which can make a system more tolerant to variations in host/DMA controller response times. AFIFO has independent FIFO buffers for read and write operations. Below are some guidelines on how to configure the AFIFO. For more information, see the device-specific TRM.

- When enabling the FIFO, you must set WNUMDMA equal to the number of transmit serializers in use. You set RNUMDMA equal to the number of receive serializers in use.
- WNUMEVT should be a multiple of the number of transmit serializers. RNUMEVT should be a multiple of the number of receive serializers.
- WNUMEVT/RNUMEVT should be configured large enough to allow for the CPU/DMA to burst many samples at once, though it should be small enough so as to avoid over/underflows. Often these are configured equal to half the FIFO depth. For example, for a 64-word FIFO, it is common to configure WNUMEVT=RNUMEVT=32.
- The DMA needs to be configured such that in response to an AFIFO event it transfers WNUMEVT/RNUMEVT words. A common configuration for devices with EDMA would be to configure the transfer as AB-synchronized with ACNT=4 and BCNT=WNUMEVT (or RNUMEVT).

## *5.4 Common Programming Issues*

The below items have to do with configuration of the McASP. All the relevant details may be found in the TRM, but it may prove useful to highlight some common issues.

### 5.4.1 You Did Not Read Back From GBLCTL After Writing to GBLCTL

The TRM’s section on initializing McASP details the various bits that need to be cleared and written in the GBLCTL register. During this process, you are instructed to poll the GBLCTL register to make sure that the writes have taken effect. A common mistake is to instead poll the GBLCTL’s alias registers, GBLCTLX and GBLCTLR (XGBLCTL and RGBLCTL in older documentation). This is not reliable. You *must* poll the GBLCTL register itself when instructed to.

GBLCTL should be read back until the bits that were written are successfully latched. This is important, because the transmitter and receiver state machines run off of the respective bit clocks, which are typically tens to hundreds of times slower than the DSP's internal bus clock. Therefore, it takes many cycles between when the DSP writes to GBLCTL (or RGBLCTL and XGBLCTL) and when the McASP actually recognizes the write operation. If this step is skipped, then the McASP may never see the reset bits in GBLCTL get asserted and de-asserted, resulting in an uninitialized McASP.

### 5.4.2 Transmitter Underrun

When the McASP underflows (underruns), McASP stops outputting data altogether (it clocks out zeroes). The only way to make it output data again is by resetting the McASP. This condition is fairly easy to detect by looking at the XSTAT.XUNDRN field. McASP can be configured to fire off an interrupt when this occurs by appropriately setting the XINTCTL.XUNDRN bitfield.

If you are not getting any data out of the McASP, you should carefully examine your initialization procedure. If you do not precisely follow the startup procedure, it is easy to underflow right at the beginning, which makes it seem as if the McASP never started up.

### 5.4.3 You are Writing/Reading to/from the Wrong Internal Port

There are two different ways to supply data to the McASP:

- Config port
- DMA port
  - This port, despite its name, is allowed to be written to with CPU or DMA.
  - The address of this port is found in the device-specific memory map. It is often named “McASP0 Data.”

Data must be written to or read from the address consistent with the programming of XBUSEL/RBUSEL. For more information, see the device-specific TRM. This bitfield determines whether reads from or writes to the serializer buffer originate from the peripheral configuration port or the DMA port. When XFMT.XBUSEL=1 (or RFMT.RBUSEL=1), the config port address should be used. For XBUSEL=0 (or RBUSEL=0), the DMA port address should be used.

### 5.4.4 You Have 16-Bit Data and are Attempting to Make 16-Bit Accesses

Strictly speaking, McASP accesses should always be 32-bit accesses. This makes it tricky if you are dealing with 16-bit data. There are a few ways to handle this:

- Increase your buffer size such that all of your 16-bit data consumes a 32-bit word (turn your array of uint16_t data into a uint32_t array with padding on each word).
- If using DMA, set your element size to 4 bytes, but your data increment to only 2 bytes. This way, you are always doing a 32-bit access, where half of the data is being discarded. You may need to use XROT/RROT=16 bits in order to get the data into the proper half of the shift register.
- You can enable the Audio FIFO, but this comes with some caveats.
  - This is not officially supported (16-bit accesses are "undefined" for the McASP), but it is widely used in any case.
  - You cannot perform "packed accesses." For example, if you are using EDMA, you need to make sure that the indexing on the McASP FIFO is set to 0, otherwise the EDMA will pack data together, which will not work.
  - When enabling the FIFO, you must set WNUMDMA equal to the number of transmit serializers in use. You set RNUMDMA equal to the number of receive serializers in use. The NUMEVT field should be a multiple of the number of corresponding serializers.
  - You may need to rotate data by 16 bits in order to get the data into the proper half of the shift register (XROT/RROT = 16-bit).