# USB-AIx-xx-xxxx / CSA_ARC: Advanced Reconnaissance Corporation

Rev 2 (wider frequency support, initial condition deterministic)

The USB-AI family's Customer Specific Application Programming Interface for ARC is designed to allow ARC to enable a new, on-card, intelligent data acquisition, analysis, and control application.

The application generates and synchronizes a pair of square waves with a 5 Hz triangle wave, with an ARC-definable phase relationship between the two square waves and the peak of the triangle wave.

By enabling the CSA_ARC feature, the normal operation of the device is superceded. Attempting to use non-CSA_ARC functions while the feature is enabled may have unintended effects, and should be vigorously avoided.

While the feature is enabled several changes to the device configuration occur.

- Digital I/O bits 0-7 are forced into output mode. Digital Bits 8-15 are forced into input mode.
- Digital output bit 1 generates a 10Hz square wave, and bit 2 generates a 50Hz square wave. Bit 0, and bits 3-7, output zero volts.
- Analog input channel zero is forced into a ±5V range, single-ended. The remaining channels can be configured as desired using a normal ADC_SetConfig statement.
- All normal ADC related functionality is disabled. Instead, all 16 channels are acquired every 10Hz, and channel 0 is acquired on a schedule designed to optimize the ability to sync the triangle wave to the digital outputs.
- ADC Oversampling is forced to zero for channels 1-15

To operate the feature successfully you should perform the following steps, in the listed order:

1. Configure: Set the desired configuration. Certain settings will be overwritten as described above.
2. Calibrate: Calibrate the device.
3. Adjust: Set the desired phase relationship between the 5Hz triangle and the 10Hz square wave.
4. Enable: Turn on the CSA_ARC feature.
5. (Optional) use the digital inputs, DAC outputs, read the ADC, and/or make further Adjustments
6. Repeat at 5 until done.
7. Disable: Turn off the CSA_ARC feature to disable the digital outputs, and return full board control to the user.

Here's some code fragments for those steps:

Configure:

```
DIO_Configure(...);
DAC_SetRange(...);
ADC_SetConfig(...);
```

Calibrate:

```
ADC_SetCal(...);
```

Adjust:

```
UInt8 p;
UInt32 pSize = sizeof(p);
GenericVendorWrite( DeviceIndex, 0xC2, 0, n, &pSize, &p);
```

n = large phase adjust, see below.
p = fine phase adjust, see below.

Enable:

```
GenericVendorWrite( DeviceIndex, 0xC2, 0, 1, 0, nil);
```

Optional:

Use DIO_Readxxx() to read the digital inputs.
Use DAC_Direct() to control the analog outputs.
Use the following code to read the ADC channels:

```
    UInt16 buf[16];
    UInt32 bufSize =32; //  sizeof(buf); (only 32 bytes is valid in order to retrieve the
ADC values)
    GenericVendorRead( DeviceIndex, 0xC2, 0, n, &bufSize, buf);
```

Disable:

```
    GenericVendorWrite( DeviceIndex, 0xC2, 0, 0, 0, nil);
```

The algorithm originally requested involved passing a voltage comparison threshold into the firmware. The firmware would have used this threshold to detect the moments, after the peak or trough of the triangle wave, when the threshold was reached, and synchronize the rising edges of both digital output square waves to these moments.

Through analysis prior to approval of the project it was determined the same goals could be accomplished by passing a delay-from-peak into the firmware. The firmware would then detect the peak or trough, instead of the threshold which followed, and synchronize to the delay after that peak or trough.

Through trial and error it was determined to be virtually impossible to detect the peaks and troughs with any reasonable accuracy using empirical triangle waves. (The signal generator in the Software department has more than 2% noise, and isn't very accurate.)

The algorithm was then redesigned to detect NOT the peaks and trough, but simply the zero-crossing. The zero crossing is 90° out of phase with the peaks of the triangle wave, and the desired digital output patterns repeat at a convenient multiple.

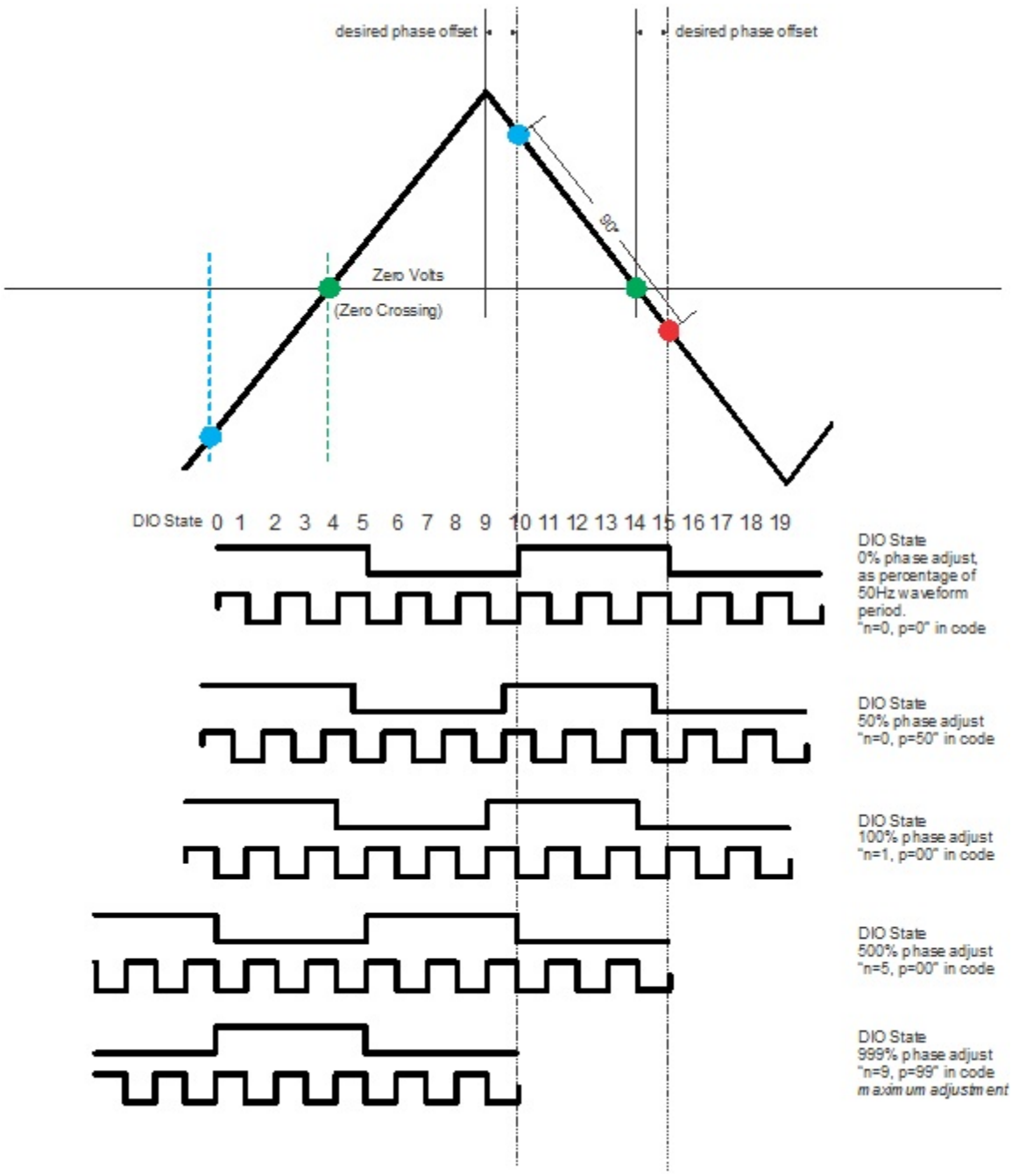
The code now has two phases:

Not Synced: The firmware reads CH0 very fast (>250 readings in ~500 microseconds), as often as practical, and determines if the zero crossing was inside the buffer thus generated. If not, it repeats, otherwise it analyses the data. Large amounts of averaging and other digital signal conditioning occur to ensure the analysis is stable. It then schedules on which internal tick the next read of CH0 should occur and switches to Synced mode, and starts the digital output pattern generator. (Which is thereby synced)

Synced: The firmware reads channels 0-15 once each and stores the data in a buffer you can read from software. It then waits for the scheduled moment, and performs the same operation as Not Synced.

As a result of the internal tick rate chosen all operations happen on multiples of 100 microseconds (10KHz).

The phase adjustment uses the digital waveform generator's state machine, and therefore has two components: a large phase adjustment by selecting to which State of the waveform the signal should sync, and a fine phase adjustment within one period of the 50Hz (faster) digital I/O waveform. See the following drawing:




Calibration for different frequencies of the triangle wave was added:

```
GenericVendorWrite( DeviceIndex, 0xC2, 0, 2, 0, null ); // start calibration
GenericVendorWrite( DeviceIndex, 0xC2, 0, 3, 0, null ); // abort calibration
GenericVendorRead( DeviceIndex, 0xC2, 0, 0, {2 through 6}, buf ); // read calibration
status and data (asynchronous)
GenericVendorRead( DeviceIndex, 0xC2, 0, 1, {2 through 6}, buf ); // START calibration,
wait for it to finish or fail, then read calibration status and data (synchronous cal)
(cal can take more than 2 seconds so we recommend using the start/status version
(asynchronous))
GenericVendorWrite (DeviceIndex, 0xC2, 0, 0, 2, buf ); write *(word*)(buf) into timer.
Warning: 0x02FF < buf < 0xFFFF.  If you put too small a value the firmware will hang.
Note: 0x04AF == 5Hz (10KHz IRQs). 0x0552 = 4.4Hz, 0x423 = 5.6Hz
```