Information Zefiro ZA2

9. Juli 1997

Inhaltsverzeichnis

1 ZA2 Programming Information

Ok, now for some programming information.... first, the program needs to know the base address of the ZA1/2 which can be 16bit I/O port 0x210, 0x250, 0x310 or 0x350...

On both the ZA1 and ZA2 the `base` address is used as a 16bit audio data port...read from it to grab the last sample, write to it to send out a sample... Of course if DMA is running, you don't need to do this.

`Base+2` is the command or status register depending on if you read or write to it. Here's what the status register looks like if you `inport(base+2)`:

bit	function	description
0	DATPCRDY	ZA1/2 goes low when a new sample is waiting at base.
1	PCDATRDY	ZA1/2 goes low when the last sample has gone out and the output buffer is ready for a new sample Note: when using DMA, these bits act as the DRQ lines to request a transaction.
2	VALID	ZA1/2 SPDIF valid sample bit read from 8412 chip.
3	FSYNC	ZA1/2 delayed 1/2 cycle, this tells if the last sample read was from the Left or Right channel
4	FREQ0	ZA1/2 frequency reporting info from 8412
5	FREQ1	ZA1/2 frequency reporting info from 8412
6	C	ZA1/2 Channel status bit from 8412, updated each sample
7	U	ZA1/2 User bit from 8412, updated each sample
8	CBL	ZA1/2 Cbit block start, 1 during first 32 Cbits, 0 during last 160 Cbits.
9	REQ	ZA2 only. REQuest line from DSP telling the PC that the DSP wants to send data to the PC...
10	PCHOSTRDY	ZA2 only. low when new data can be written to host port on DSP... better to use FSYNC to sync then this..
11	FREQ2	ZA2 only. yet one last bit for frequency reporting.

Ok! now the command port...when you write to `base+2`...`outport(base+2,cmd)`

bit	function	description
0	TOS/COAX	0 sets toslink input
		1 sets coaxial input
1	AES/EBU	1 sets AES/EBU input
		0 sets either coax or toslink.
NOTE! bits 2-7 are different from the ZA1!!!!!!		
2	IRQSEL0	00 sets interrupts disabled
3	IRQSEL1	01 sets IRQ 10
		10 sets IRQ 11
		11 sets IRQ 12
4	IDMASEL0	00 sets input DMA disabled
5	IDMASEL1	01 sets input DMA 5
		10 sets input DMA 6
		11 sets input DMA 7
6	ODMASEL0	00 sets output DMA disabled
7	ODMASEL1	01 sets output DMA 5
		10 sets output DMA 6
		11 sets output DMA 7
8	MODE0	00 mode 0: uses the 48khz clock for everything.. this is used during initial upload
9	MODE1	01 mode 1: sets the ZA2 for standalone output with possible feedback to the PC.. this is used by PLAYDAT or anything needing a stand alone clock source for output. PC→DSP (→PC)
		10 mode 2: sets the ZA2 for input only or externally clocked input and output.. this is basically the ZA1 mode. INPUT→PC (→DSP)
		11 mode 3: This is the pass thru mode.. it allows the input signal to go directly to the DSP, then the DSP sends 16 bit data to the data register (`base`) rather then the 8412 input chip doing so... INPUT→DSP (→PC)

Here's how some of the signals are routed during these modes...

MODE	internal clock from.	clock to DSP.	Passthru?	data input source.
0	6.144mhz clock	6.144mhz	no	DSP (disabled)
1	clkout DSP PLL	6.144mhz	no	DSP
2	8412 input chip	8412 MCK	no	8412 input chip
3	clkout DSP PLL	8412 MCK	YES	DSP

anyway, back to the command register

bit 10	WCSEL	Word clock select (for future use)
bit 11	PIO	sets PIO on DSP and external connector (future use)
bit 12	RESET	tied directly to the reset pin on the DSP
bit 13	CS	tied directly to the CS pin on the DSP
bit 14	BOOT	tied directly to the BOOT pin on the DSP.

thats all...

`base+4` is the host output port.. lower 8 bits are sent to the DSP when in DSP read mode... when the DSP host port is in write mode, the data is assembled in 16bit chunks and sent to the data register (base).

`base+6` is used to program the xilinx chip in upbit.exe don't mess with this because you could fry the chip.. that would be bad.

2 ZA2 Modes

The ZA2 modes are 0 thru 3...and are set by bits 8 and 9 in the command register (`base+2`)...so for example mode 3 would be — 0x300. Here's a rundown on the hardware config for the 4 modes:

'internal clock' is the master clock that the glue logic runs at...This is a 128xFs clock...from this 'Master' the serial bit clock (SCK or SCLK) and the Frame sync (FSYNC) are generated (64x and 1x). this is the speed at which data is transfered to or from the computer. The ZA2 cannot input and output at different speeds, input and output are locked together.... if we record from the outside world, our clock must be derived from the SPDIF signal (as the ZA1 does)...if we're playing stand alone 44.1Khz (for example) then the internal clock must be 128x44.1khz.

'MCK to 4920' is the clock signal sent TO the DSP chip as the master. If we want to do simultaneous record/play, we must be synced to the outside world...if we want to do live sample rate conversion we must be locked to the outside world, if we do stand alone playback, we derive our clocks from the 6.144Mhz oscillator on the board.... note the DSP can synthesize any frequency from this 6.144Mhz master and then SEND IT BACK TO THE XILINX! for example, if we want 44.1khz stand alone playback (mode 1) we set the DSP to use 6.144 as the master, SYNTHESIZE 44.1 (5.6448Mhz) inside the DSP then send this 5.6448Mhz signal to the Xilinx chip for it's "internal clock"

'pass-thru' normally the 16bit digital audio input to the DSP comes from the Xilinx (ie: from the PC) but in a few cases we want to bypass the PC and have the spdif input go direct to the DSP...this is the pass thru mode. Note that for 48→44.1 conversion, the ZA2 is in pass thru mode, but is deriving the DSP clock from the input rather then the 6.144Mhz crystal...this is needed since we must be synced to the input (otherwise our 48khz and the outside 48khz might not match)...the audio is sent at 48khz directly from the 8412 spdif input chip directly to the DSP...the DSP then synthesizes 44.1khz from this...it reclocks to 44.1 and does all the math so that the samples come out correct (when running 48244...if you just put it in 44.1 mode with 48 input then some samples will be repeated without the 48244 program). note also, in this mode the Xilinx is clocked at 44.1...a 5.6448Mhz signal derived from the input rather then from the 6.144mhz crystal...if you run 48244 and your input is 32khz, you'll get a nicely resampled 29.4Khz signal. Note also that unlike modes 0, 1 and 2 where FSYNC and SCK are internally generated, in mode 3 there are 2 FSYNCS and 2 SCKS...one from the 8412 acting as

master to send data to the DSP, and the other derived from the DSP to get data from the DSP to the PC.... this is how we move 48khz data to the DSP and read 44.1 from it at the same time.

'Source'... this is what is feeding the input to the PC/Xilinx... we can either get our 16 bit audio from the CS8412 (normal input mode... just like a ZA1) or we can try to squeeze 16bit audio out of the HOST port on the DSP... this is how 48244 works... the signal goes INPUT→DSP→PC.

ok, here's the rundown:

- MODE 0: config mode.... made for initial upload to the DSP.
  Internal clock: 6.144Mhz
  MCK to 4920: 6.144Mhz
  Pass-thru: NO
  Source: N/A.
- MODE 1: Playback mode... used for stand alone play or play with DSP feedback.
  Internal clock: Clkout (from DSP)
  MCK to 4920: 6.144Mhz
  Pass-thru: NO
  Source: DSP (host port)
- MODE 2: ZA1 mode... used for recording or simultaneous record/play.
  Internal clock: from CS8412
  MCK to 4920: from CS8412
  Pass-thru: NO
  Source: CS8412
- MODE 3: Pass thru mode... sets the ZA2 to INPUT→DSP→PC mode rather then the default INPUT→PC→DSP mode. Internal clock: Clkout (from DSP)
  MCK to 4920: from CS8412
  Pass-thru: YES...
  Source: DSP (host port)

That's about it... if you have questions on this, let me know and I'll clarify.

3 The DSP Operating System!

6/3/95

Ok.. here's some info on the DSP OS.... When the ZA2 comes alive using the UPBIT program, the DSP is still somewhat dead... actually it's being held in reset because the default of bit 12 (reset) in the command register is zero. The UPSIMZ program will upload a .SIM (DSP executable in HEX) file to the DSP and start it up... the DSP has a small bootstrap program built in. This bootstrap handles the upload process and after checking that xfer went ok, performs a software reset and begins executing the program. One of the first things the DSP program must do is set up the PLL circuits on the chip to the desired frequency (when the chip first comes up, it's running in a slow mode). The result of this PLL divide/multiply ratio feeds all parts of the system including the DAC, SPDIF output, DSP clock and in modes 1 and 3, the ISA bus interface.

What program should we load into the DSP? Well most of the time you'll probably use the 'operating system' program... currently TZ1.ASM. Another program will be the 48 to 44.1 conversion program, and eventually an MPEG audio decoder and whatever other goodies we can come up with. For now though, all the DOS software (and future windows software) will expect the DSP to react to a particular protocol set up by the OS program TZ1.ASM. The OS program runs entirely on the DSP's interrupts, the main loop does nothing. There are 7 interrupt routines in the OS: left channel input, right channel input, left channel output, right channel output, host port input, host port output, and the long interrupt (ISR8) which handles the channel status bit block update and takes over if the PLL breaks lock. There's a lot more going on in these 7 subroutines then you might expect. Since the Left in, Right in, Left out and Right out all get called at the current sample rate (48000, 44100, 32000 or whatever times per second) ... any one of these would be a good place to put periodic routines like updating the user bits (for Start_IDs etc), forcing the ZA2 interrupt, syncing the input L/R or syncing the output L/R... The volume is adjusted on the output side (Right out and Left out). There is also a 'Resync' routine nestled into the Left input routine... this relocks the channel status block and a few other oddities when needed.

Ok, now the tricky part.. the HOST interface. The DSP has a built in serial 8 bit host port that can only move data in one direction at a time. A special pin on the chip and a special sequence of bits must be sent to the DSP in order to select input mode or output mode. When the DSP first comes up, it is in 'input' mode so that the user can upload

the program to the DSP (by writing to base+4 with 8 bit information). I've designed the ZA2 so that when this host port is in 'output' mode, the data can be shuffled directly into the audio register of the ISA interface.. this way 16bit audio can be read FROM the DSP to the computer. The DSP is set up to be interrupted each time 8 bits of information are sent out, so we load half of the 16bit word into the SCPOUT register then set a flag so the system knows that the next byte sent will be the second half of the 16bit audio sample.. on the ISA side all 16bits are shuffled in transparently... the problem is getting this whole thing synchronized.. At first I was thinking of syncing to FSYNC so that FSYNC (bit 3 in the status register) would indicate left or right. But this has always been a hassle (on the ZA1 this required special timing to make sure DMAs were started on the correct sample.. even needed a VxD on the windows drivers to handle this). Soooo... now we sync up by sending a special command to the DSP, then waiting for a signal mixed in with the audio that identifies the left channel.. there's also a similar sync procedure for sending audio to the DSP. The problem with this host port is that it's difficult to tell if it's in 'input' or 'output' mode.. The bit clock used on the host port is the same clock that's used to move normal 16bit audio thru the system. Unlike SCK for the normal audio, the bit clock for the host port must only be clocked 8 times per byte. If the DSP is in 'input' mode then 8 bits get shuffled in and an internal interrupt goes off to let the DSP know a byte has arrived. If the DSP is in output mode, the last byte written (zero if none) gets clocked out and an interrupt tells the DSP that the output buffer is clear. Note that if the DSP writes a byte to be sent out and the computer DOES NOT clock the port 8 times to read it, then the REQ line goes low indicating to the host (the PC) that the DSP has something to send it... If we are sending audio from the DSP to the PC ('output' mode) then the REQ line is always low because there is always a byte pending.

Now the hardware on the ZA2 has no idea if the DSP is in 'input' mode or 'output' mode.. so here's how it works: If you write a word to base+4, then the port is clocked 8 times and the lower 8 bits are sent to the DSP... hopefully the DSP is in 'input' mode, otherwise you've clocked 8 bits out of the DSP and into the bit bucket. If the ZA2 is in mode 1 or 3 and the REQ signal is low, then at the next FSYNC transition, 16 clocks are sent to the DSP to bring in the next audio sample (which on the DSP side is actually 2 transactions). If REQ is low and the ZA2 is in modes 1 or 3 and the host port is in 'input mode' then the ZA2 clocks 16 bits (2 bytes) out of a bit bucket somewhere and into the DSP host input register (SCPIN).

The DSP 'OS' was written so that a special protocol is used to send commands to the DSP and set up the DSP to be read from. When the 'OS' first comes alive out of reset (after UPSIMZ) the REQ line is high indicating no pending transaction.. it stays like this until the DSP writes something to the SCPOUT register. The OS waits a few cycles before sending anything to the host output so REQ should stay high for a few clock cycles. The bootstrap program expects a 24bit checksum after uploading (in the UPSIMZ program) and if this checksum is bad, then REQ goes low right after the download. UPSIMZ looks for this signal right after the checksum is sent to make sure

all went well. When the DSP finally does pull REQ low ligitimately it starts by sending 0x7a61 'za' about 38 times, after that true audio should follow starting with the left sample. Most of the time you are not concerned about this so you can just ignore it. But if you are setting up to receive info from the DSP, then you should watch some of these 0x7a61's to make sure the host output was synchronized correctly. This seems to happen about 80% or 90% of the time.. if it doesn't work you'll need to either reset the DSP or send a command to it (switching directions on the host port 2x) in order to re-synchronize.

There are 3 subroutines in the "setzaenv.c" file used to set up the direction on the host port for input or output and to send commands to the DSP. We always assume that after the DSP OS has come online, that the REQ line is low no matter what mode you're in.. even if you haven't put the DSP in 'output' mode, the OS will try to send the 0x7a61 sync about 20 cycles after coming alive, thus setting REQ low...it will stay low because new data gets put into the port even if the ZA2 is reading it as fast as possible. The `OpenDSP()` routine is used to set the DSP host port to 'input' mode. The `SendDSPCMD()` routine is used to send a command to the DSP.. the command consists of an 8 bit command and 24 bits of data.. the OS expects all commands to be in this form (1+3). To send a command to the DSP you must first use `OpenDSP()`, then send as many commands as you want with `SendDSPCMD()`.... keep in mind that during this time, the ZA2 is held in either mode 0 or mode 2 so that the ZA2 doesn't try to read 16bits from the DSP while it is in 'input' mode. When you are done, you should use `CloseDSP()` to put the host port back in 'output' mode. CloseDSP also checks to see that the REQ line goes high, then eventually goes low again as it should after a transaction with the OS. `CloseDSP()` also checks for the 0x7a61 'za' sync words and will return a zero if the sync works.. if not it will return either a 1 or 2. The sync is only important if you're intending to read from the DSP.

The commands sent to the DSP are all in the form 8 bit command + 24 bit data. Sending a zero to the DSP will be ignored. Sending a 0xff to the DSP will cause it to try and resync...this is done in `CloseDSP()` right before switching to 'output' mode. All commands (at present) have bit 7 set, if the DSP gets a byte with bit 7 set (that is not 0xff) then it waits for the next 3 bytes and assembles them into a 24bit word. The lower 7 bits (0 thru 6) of the command byte are used as an offset into the data ram of the DSP (all system variables are stored here)...the data is then written to that location. For example command=0x80 and data=0x000000 would write a zero into the first location of data ram...command=0x81 would write into the second location in ram...Here are some of the addresses used at the moment (in decimal):

129 Volume: the number at this location is multiplied against the output samples before going to the DAC and SPDIF out. 0x400000 is full volume and should be used at most times.

137 CM1: This is the value sent to the CM1 register for the PLL divide/multiply ratio... in other words, the sample rate. If you change this, you'll need to reset (`ResetDSP()`) for the change to take effect.

138 Lucnt: Left user bit count.. this is the clock pulse sent out from a DAT deck on a SPDIF stream every time the head makes a revolution (2000rpm) This is used to make Start_IDs. The value should be 1440 for 48khz, 1323 for 44.1 and 1920 for 32lp.

139 Rucnt: Right user bit count.. this is actually a start ID counter and should be set to 300 whenever you want to send out a Start_ID.. Start_ID is the right bit on after the left clock pulse for 300 drum rotations (at 2000rpm that's about 9 seconds).

142 Irqcnt: This is the number of samples between IRQ pulses.. haven't done much with this yet, but normally you'd set this to half your DMA buffer size.. so if the DMA buffer was 4k, this would be set to 2048... note you also need bits 2 and 3 set up in the command register to enable the appropriate IRQ line.

146 Psync: This is used to swap the channels in order to sync the output if needed.. this is used in place of FSYNC to synchronize the output.. 0 leaves the output muted until the 0x7a61 signal is found by the DSP in the audio stream... then it sets this to either 1 or 2 depending on if R→R L→L or R→L L→R. Also if you set Psync to 3, it sends out the difference of the Right and Left channel on both.

224-247 These make up the Channel Status bit block... each 24 bit address holds 8 bits of channel status for a grand total of 192. The most significant 16 bits hold the 8 Cbits.. 1 for the Right channel and one for the left... normally these are set the same.. So bit 0 in the 192 bit Cbit block would be set by bits 8 and 9, bit 1 would be 10 and 11 (with command 224). By writing to memory between 96 and 119 (commands 224-247) you can set the 192 bit Channel status block to look like whatever you want.. in SPDIF, bit 2 is SCMS and bit 3 is Emphasis for example. You can look at all these bits on the ZA1/2 input by typing verf x x.

The `SendDSPCMD()` takes an integer as the first parameter (the 7 bit command with the 8th bit set) and a long integer for the second (the 24bit data to be sent to the DSP). Remember to `CloseDSP()` when you're done to set the direction on the host port back to 'output' mode. Sometimes the sync in CloseDSP fails... this is not a big deal unless you're intending to receive data from the DSP (like for recording if you choose to route through the DSP).. in which case it can be very annoying.. Hopefully `OpenDSP()` and `CloseDSP()` should be enough to set it straight, but this doesn't always work and sometimes you actually need to reset the DSP. Resetting the DSP is less desirable because

you get a nasty click from the DAC.

The DSP resets itself in many ways whenever the PLL breaks lock... there are many ways to break lock... If you are deriving your source from one of the inputs, and the source dies (ie, you turn off the DAT deck or CD player) this will cause the DSP to break lock. Note, the DAC tries to mute when you break lock, but often there is a nasty click. When the ZA2 is in mode 0 or 1 it's sending a 6.144mhz signal to the DSP to clock from. When the ZA2 is in mode 2 or 3, it's deriving its clock from the currently selected input.. obviously switching between modes 0 or 1 and 2 or 3 will probably break lock... If you wrote a DSP program that tried to change sample rates on the fly, this would also cause a temporary break in the clock. I'm still looking for ways to avoid the clicking on the DAC, but so far haven't found any.. at least the mute works (after the click)... the way it was before, if your source died, the DACs made a terrible screeching noise (not good!).

Hmm.. that's about it for now.. as always if you think of something that needs to be clarified, let me know and I'll fix the document.. I can be reached at hanssen@netcom.com or by voice at (714)-551-5833 or (714)-551-8880.