CIRCUIT CELLAR INK®
THE COMPUTER APPLICATIONS JOURNAL

FEATURE ARTICLE

PIC'Spectrum

Robert Lacoste

Audio Spectrum Analyzer

Sure, a DSP can make the calculations and generate the pixel bitmap for an audio spectrum analyzer, but at what cost? Robert gets the same result from a single PIC processor with a design so good he walked away with a Design98 first prize.

I remember being quite astonished when one of my professors explained the basics of frequency domain analysis, "Every periodic signal is the sum of pure sinusoidal signals, with given frequencies, amplitudes, and phases."

Hmm...every signal. That includes the light coming from the sun, the vibrations of my old car, even the tears of my two-month-old baby!

And, I bet that 99% of *INK* readers are like me. You want to understand, and you understand it better if you make it yourself.

So, years ago, I quickly wire-wrapped an analog acquisition board and wrote a small BASIC program on my old Apple II to display the frequency decomposition of an incoming audio signal. It was my first audio spectrum analyzer. Later, I did the same on my PCs, and the '*x*86s were soon powerful enough to get real-time performance.

Using a PC is OK, but how about an autonomous device? Something small enough to bring along anywhere but that has a VGA video output with a decent quality image and real-time refreshes.

My first idea was something like the block diagram in Figure 1a. An amplifier and low-pass filter suppress out-of-the-band signals before the signal goes to an ADC, which transforms it into numerical samples.

A DSP can calculate the frequency decomposition of this signal with the classic Fast Fourier Transform (FFT) algorithm and generate the pixel bitmap in video RAM, which is displayed by a CRT controller chip.

You'd need a fair amount of computing power, but is it possible to do the same with a simpler design? How about a high-end microcontroller? *INK*'s Design98 contest presented a great opportunity to try it with one of the newer Microchip devices—the PIC17C756.

This chip has enough horsepower not only to do an FFT in real time but also to eliminate the CRT controller. The video output can be made with some general-purpose parallel output lines, and the software toggles the corresponding bits to generate the video

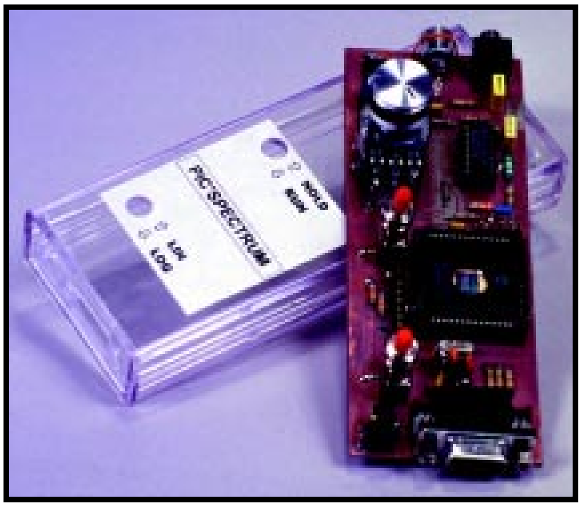


Photo 1—*The PIC'Spectrum prototype is built on a 1″ × 3″ PCB. A good ground plate is mandatory on a mixed digital/analog design like this one. I used a transparent box to prove that there are very few components inside.*

synchronization and color signals in real time.

With this controller, the block diagram of my logic analyzer (shown in Figure 1b) is, well, as simple as possible. PIC'Spectrum is born!

YOU SAY PIC17C756?

Before getting into more details about PIC'Spectrum (see Photo 1), let's have a look at the PIC17C756 and its stripped-down version, the PIC17C752. These new fully static CMOS chips are an enhancement of the existing PIC17C4X family.

The microcontroller core runs up to 33 MHz, giving a 121-ns instruction cycle. These chips are pure RISC design, and thanks to the two-stage pipeline, each instruction executes in only one cycle, except program branches and table reads/writes, which are two cycles long. That makes it near 8 MIPS—not bad for a microcontroller.

The 58 single-word instructions (coded on 16 bits) are easy to learn for any user of Microchip's smaller controllers. Direct, indirect, and relative addressing modes are supported. External interrupts are present, as is a 16-level hardware stack.

One interesting feature for compute-intensive applications: there is an integrated 8 × 8 bit hardware multiplier, working in only one instruction cycle. This multiplier offers a performance boost of more than three times (compared to a software-only version) for a complete FFT calculation.

The internal memory is impressive as well. The PIC17C756 has 16 K words of EPROM program memory as well as 902 bytes of general-purpose RAM, which accommodates quite large projects without the need of external memories, even if extended modes are available and support up to 64 K words of program memory.

The smaller PIC17C752 has only 8 K words of EPROM and 454 bytes of RAM. Microchip has also announced flash-memory versions (PIC17F75X). The memory map is shown in Figure 2.

One piece of bad news: the RAM and registers are still banked, and the working page is selected by some bits in the BSR register. Even if some specific assembler instructions help, this is a major headache for the programmer and a major source of bugs. I hope that Microchip switches to a linear address mode in the near future.

Hosted in 64- (DIP) or 68-pin packages (PLCC and TQFP), the PIC17C75X provides 50 I/O pins with individual direction control. As usual, each pin may be used for general-purpose I/O or dedicated to some on-chip peripherals.

Have a look of the pinout of the PLCC version in Figure 3 and you'll understand that this chip is quite flexible. Both OTP and windowed versions exist, even if they aren't so easy to find.

On the peripheral feature list, there are four timers (two 16 bits wide and two 8 bits wide, TMR0 having an internal 8-bit programmable prescaler), four capture input pins, and three PWM outputs with a 10-bit resolution.

Need more? Perhaps two asynchronous and one synchronous serial port, the latter configurable both in SPI and I^{2}C modes, master or slave? Or a 12-channel 10-bit ADC? Or an RC-clocked watchdog timer?

How about an integrated supply-voltage supervision? Or a configurable RC/crystal/ceramic clock system with an oscillator start-up timer? Name it, and it's probably there.

The PIC17C756X also has in-circuit programming hardware. By pulling the TEST and MCLR/V$_{PP}$ pins to a 13-V programming voltage before powering up the chip, a specific ROM bootstrap code is launched and waits for orders coming from a dedicated serial synchronous interface (RA1/RA4/RA5 pins).

With a small interface connected to the LPT port of your favorite computer, you can read or write the internal

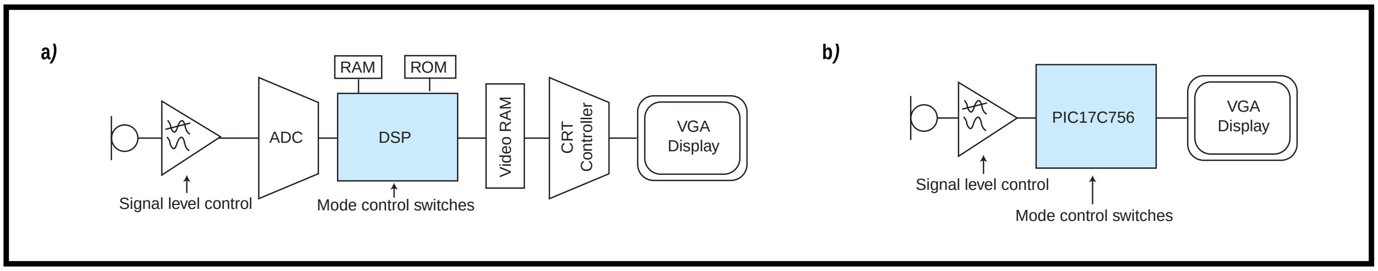


Figure 1a—*Here's a classic block diagram for a spectrum analyzer. The signal is amplified and low-pass filtered before going to an ADC. A DSP calculates the FFT and drives the VGA screen through a video controller.* **b—***The PIC'Spectrum block diagram is much simpler. The microphone signal is amplified and low-pass filtered and goes directly to the PIC, which calculates the FFT and directly generates the video. It's all in the software.*

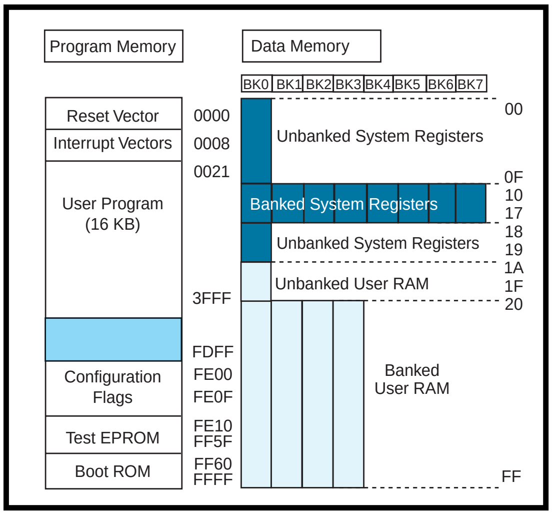


Figure 2—*From this memory map of the PIC17C756, you see that the program and data areas are separated, as usual on a Harvard architecture. The '17C752 has the same memory map, except that there is only 8 KB of user program memory and two user RAM banks.*

EPROM without any specific programmer and without having to extract the chip from its socket.

I developed my own PC-based in-circuit programmer based on Microchip's programming specifications [2].

Electrically, the standard-grade PIC17C756 needs a supply voltage from 4.5 to 6.0 V, while a special extended device (PIC17LC756) accepts anything from 3.0 to 6.0 V. In both cases, the minimum RAM retention voltage is 1.5 V.

The supply current is 6 mA at 4 MHz but climbs to 50 mA at 33 MHz. Of course, sleep modes reduce consumption down to 1 µA, depending on the selected peripherals, but high performances and low consumption are still difficult to conciliate.

PIC'SPECTRUM HARDWARE

Now that we have a good microcontroller, let's look at the PIC'Spectrum hardware shown in Figure 4.

A small power supply, which is built around U1 (78M05), generates a clean 5 V from a standard 9-VDC power supply (the total power consumption is around 100 mA, which is mainly 50 mA for the PIC and 50 mA for driving the three 75-Ω video outputs). The LED D1 indicates powerup and generates a pseudoground 1.9-V level used by the analog section.

The analog input signal (from an onboard electret measurement microphone or a line input jack) is amplified and low-pass filtered down to 10 kHz by four operational amplifiers (U2, LM324). A potentiometer lets you adapt the amplifier gain to the ambient sound level and serves as an on-off switch. The output signal, centered on the 1.9-V pseudoground level, connects directly to one of the PIC's analog inputs.

The PIC processor is clocked by a 32-MHz crystal (I wasn't able to find a 33-MHz crystal in time). As usual for these frequencies, this crystal is a 3× overtone model.

I needed a damper circuit (L1/C8) to select the correct resonant frequency. Without it, the crystal would oscillate on its fundamental frequency, and I'd end up with a 10.66-MHz clock!

Two switches (K1 and K2) control the current mode (run or hold) and the scaling of the display (linear or logarithmic). They connect directly to RB6 and RB7 because the PIC has selectable internal pullups.

The VGA-compatible video-output connector is directly driven by the PIC. The horizontal and vertical synchronization signals are TTL compatible, so there's no problem there. For the R, G, and B lines (0–2 V/75 Ω), a 150-Ω series resistor does an adequate 5-V to 2-V/75-Ω adaptation, thanks to the high power capacity of the PIC outputs (20 mA/line, 100 mA total for ports A and B).

The 8-pin header J4 handles in-circuit serial programming. Four additional output pins (including one UART output) are connected to a debug header. This provision is useful during the debug phase because it enables you to send debug information to a serial terminal to find out what's happening inside the box when you don't have an in-circuit emulator.

ON THE SOFTWARE FRONT

Of course, when you choose the simplest possible hardware, the software has more to do. Here, the software must:

- do the acquisition of a burst of the analog signal (typically 256 samples at a sampling frequency of 16 kHz)
- calculate the FFT of these 256 samples, giving an amplitude for each of the 128 frequencies
- calculate the power of each frequency and scale it (depending on the position of the log/lin switch)
- manage the run/hold switch
- do the generation of the VGA video signal in parallel

To do all these tasks while keeping real-time requirements, PIC'Spectrum software is divided into two tasks. The

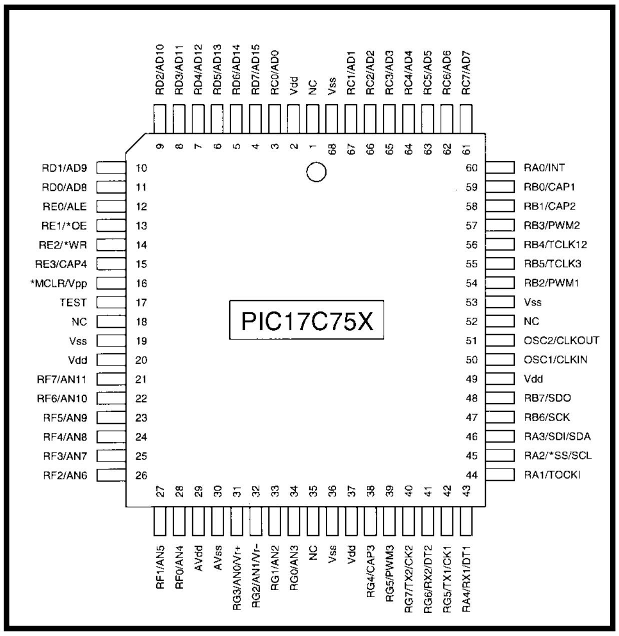


Figure 3—*The pinout of the PIC17C75X (LCC68 package, top view) gives you a good idea of the number of on-chip peripherals.*

main program is in charge of initialization, device control, and numerical algorithms. The interrupt routine, executed each 31.77 µs (corresponding to the VGA horizontal video synchronization period), is in charge of analog signal acquisition and video generation.

These two tasks dialog through three RAM shared structures:

- FFT buffer (256 words, 16 bits each), filled by the interrupt routine with analog samples and used by the main program for FFT calculation
- display buffer (128 bytes), filled by the main program with the length (in pixels) of each of the 128 horizontal frequency bars, and used by the interrupt routine for video generation
- synchronization variables

Figure 5 illustrates the information flow. Since you can get the complete source code from the Circuit Cellar Web site, I'll focus here under on the more specific codes, like FFT calculation and software video generation.

A standard FFT algorithm works

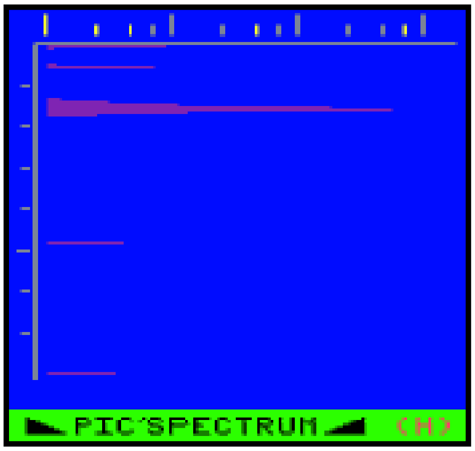


Photo 2—*This video image was generated by PIC'Spectrum. Here, we have a quite clean 1.6-kHz signal. The third and fifth harmonics are visible under the fundamental frequency.*

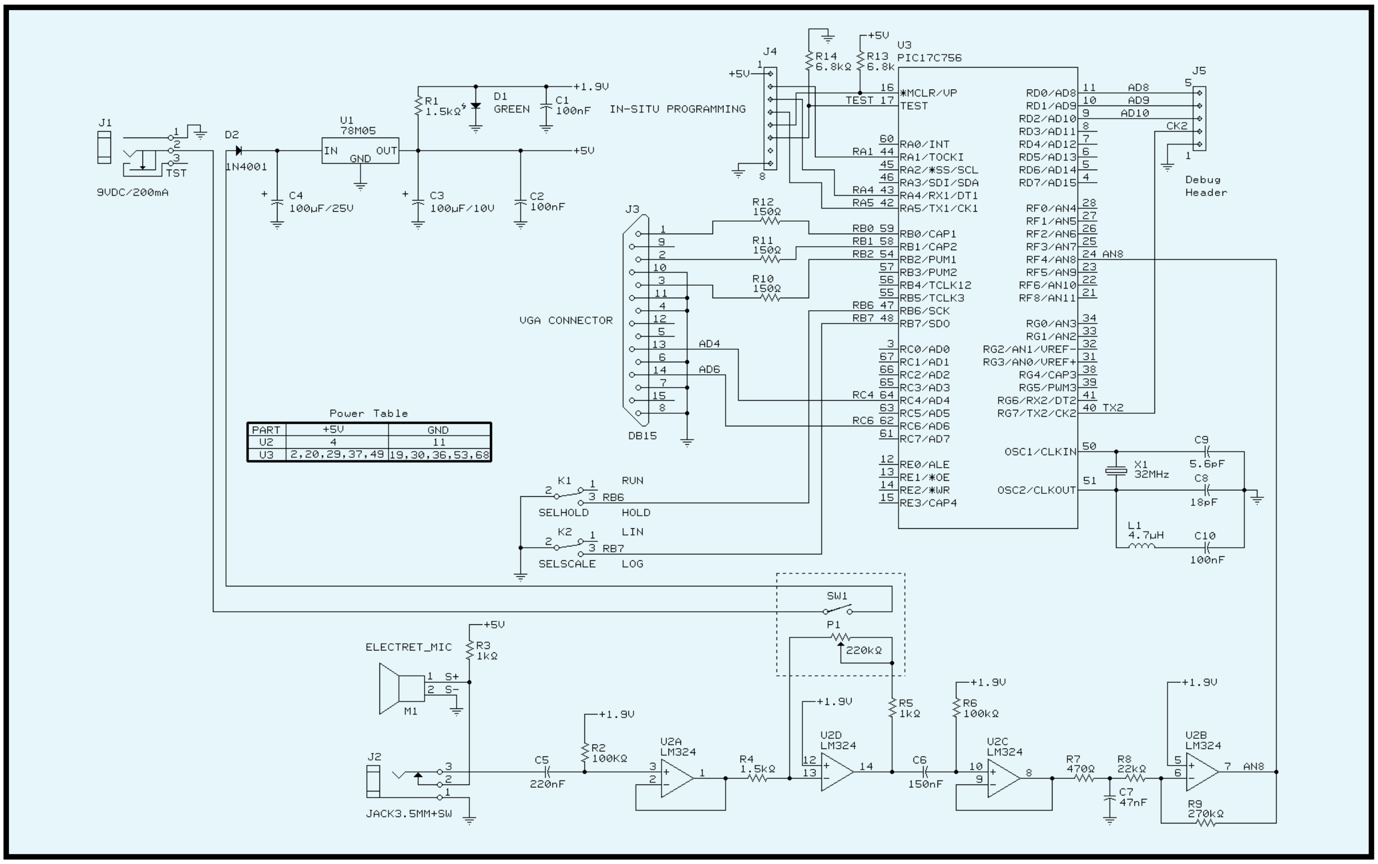
on complex numbers (real plus imaginary parts). It takes as an input an array of *n* complex samples and gives an array of *n* complex frequency amplitudes. Here, the input signal is of course only real numbers.

The immediate solution to get its FFT is to add a zero imaginary part to each sample and to use the standard complex numbers algorithm. If you do that, only half of the *n* frequencies in the result are useful (in fact, each value is found twice).

This is a consequence of the well-known Nyquist rule. If you sample your signal at a period of 1/*n*, you can only analyze frequencies with periods above 2/*n*. More problematic, this simple approach uses twice as much memory as is useful and we have only 902 bytes of RAM.

Fortunately, you can use a more sophisticated algorithm, known as real-mode FFT. The idea is to pack two real samples in each complex input value, do a standard complex FFT, and decrypt the resulting complex values to find back the good real figures. You can get the details from *Numerical Recipes in C: The Art of Scientific Computing* [3].

To implement this FFT algorithm on the PIC, I developed a fixed-point mathematical library (source file is `fixed.inc`). It implements a virtual fixed-point machine operating on two



PART	+5V	GND
U2	4	11
U3	2,20,29,37,49	19,30,36,53,68

Figure 4—*Around the PIC, there is only a quad operational amplifier, a regulator, and some discrete components.*

16-bit floating-point registers (RA and RB).

Routines are available for addition, subtraction, multiplication, division, sine, and logarithm, as well as access to the banked RAM. The fixed-point format used is S/2/13 (one sign bit, 2 bits for the integer part, and 13 bits for the fractional part). In fact, writing sinus and logarithm calculation routines in assembler is quite an interesting experience.

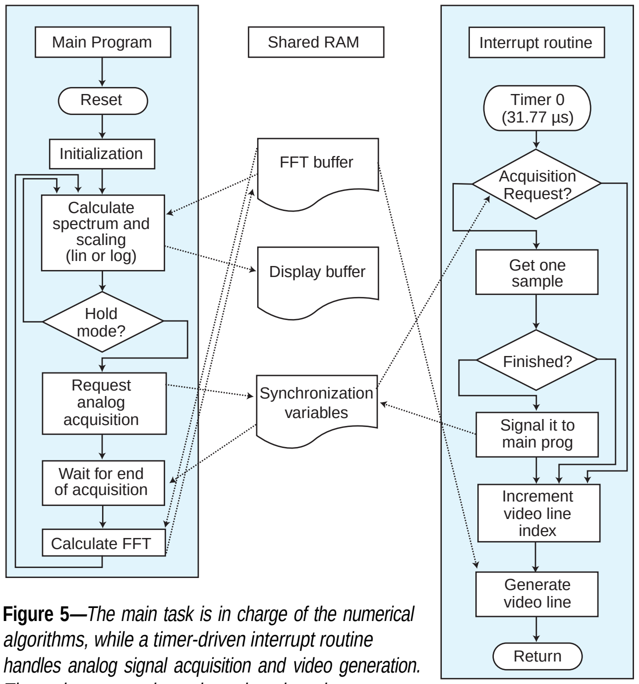


Figure 5—*The main task is in charge of the numerical algorithms, while a timer-driven interrupt routine handles analog signal acquisition and video generation. The tasks communicate through a shared memory.*

OPTIMIZING VIDEO INTERRUPTS

Writing the interrupt routine was another interesting exercise. One interrupt is generated by timer 0 each 31.77 µs. This period equals the horizontal refresh period of a VGA signal in mode 7 (640 × 350).

Because 31.77 µs translates into 254 instructions only (even at 32 MHz), the number of instructions used to do the analog signal acquisition and the video generation must be carefully optimized.

Figure 6 shows the VGA horizontal timing specifications, the corresponding number of instructions that the PIC17C756 could execute at 32 MHz, and the operations done by the interrupt routine in the corresponding time frame (more information on VGA timings is available on the Web [4]).

The interrupt routine starts with housekeeping (register saving, timer reload, etc.) and switches on the horizontal synchronization signal. During the 3.7 µs needed by this synchronization, the software manages the acquisition of an analog sample (one signal sample is taken every two interrupts, giving a 15.7-kHz digitization rate).

The current video line number is then incremented and used as an index into a table, giving the address of the routine to use for displaying each scan line (vertical synchronization virtual lines, blank lines, frequency display line, scale or title display, etc.). Each routine manages the switching on and off of the three RGB video signals with good timing.

This brute-force programming method is needed to achieve the result in Photo 2. The longer the program takes between the interrupt and the start of the video display, the more black areas you have onscreen.

Real-time requirements have to be managed very carefully. In particular, the execution time from the interrupt to the start of the video display must be rigorously constant, whatever the results of the if/then tests are.

If it isn’t, the video lines do not align. I was forced to add NOPs everywhere—in particular, in the analog acquisition routines—to ensure that the same number of instructions is executed whatever the situation is.

To get the best result, I implemented some strange programming practices. For example, to get a specific delay with 125-ns resolution, I do a calculated jump in the middle of a long sequence of NOPs. It’s the only solution I’ve found, because managing a loop consumes several instructions.

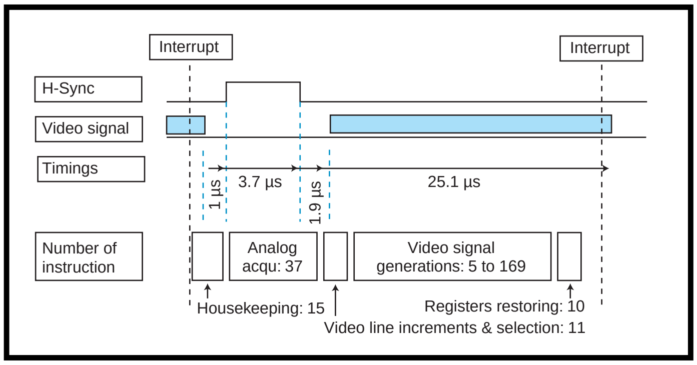


Figure 6—*The top part shows the timings of the video and horizontal synchronization signals, while the bottom part shows the conversion of these timings in number of instructions as well as what the PIC is doing during this time.*

DEVELOPMENT AND DEBUG

Debugging a signal-processing embedded application isn’t easy. And, I don’t have access to an adequate ICE. Since it might be useful for similar projects, let me briefly explain my methodology (which must not be too bad: PIC’Spectrum actually works!).

After some timing calculations to ensure the feasibility of the concept, I started with a prototyping phase on a PC using a PC sound board as input. I used floating-point calculations, with everything in C (see Listing 1). I then translated the software to use exclusively integer numbers and debugged it.

The third step was to develop a fixed-point library (still on the PC) that prefigures the future PIC-based fixed-point library. I also modified the FFT code to do all calculations exclusively through this library. It was quite easy to write this fixed-point library in PIC assembler, debug it on Microchip's simulator, and translate the FFT routine from C to assembler.

Using the simulator is useful for numerical-calculation software. In fact, it was possible to open two windows on my PC, the first being a standard PC debugger with my C-code FFT, and the second the PIC simulator with the manually translated assembler code. Single stepping between the two codes with a test signal as an input helped me quickly find the more tricky bugs.

When the signal-processing part was completed and debugged, I wrote the real-time part (signal acquisition and video generation) and debugged it as much as I could on the simulator.

This phase was helpful for correcting timing issues related to video synchronization. For example, with a breakpoint set on each access to the horizontal synchronization pin, it's easy to verify if the timings on this pin meet the VGA specification.

I waited until this phase was successful before cabling the prototype. The first EPROM I burned didn't work 100%, but I got a working video image and something resembling a spectrum display.

Thanks to the in-circuit programming and some debug pins, the software was working soon. In fact, 90% of the bugs found in this last step were related to banking register issues, which are tough to simulate because hardware ports are involved.

Listing 1—*Here you see my step-by-step process from traditional float C code on a PC down to integer assembler code on a PIC.*

```
Step 1: C code on PC, float numbers:
float data[NMAX],h1r;
…
data[j]=(data[i]-h1r)/2;    /* calculation step of FFT routine */

Step 2: C code on PC, integer numbers:
word data[NMAX], h1r;
…
data[j]=(data[i]-h1r)/2;

Step 3: C code on PC, use of pseudoregister based on fixed-point
        library:
fixed.h:
  word ra, rb;              /* pseudoregisters A and B */
  …
  m_sub ()                  /* subtraction function a-b->a */
  {ra=ra-rb;};
  …
main program:
  …
  /* data[j]=(data[i]-h1r)/2 */
  m_ldai(i);  /* load data[i] in A register */
  m_ldb(h1r); /* load h1r in B register */
  m_sub();    /* A = A-B */
  m_div2();   /* A = A/2 */
  m_stai(j);  /* store A in data[j] */

Step 4: Last, translation in PIC assembler
Fixed.inc:
  …
  ; m_sub :   subtract B to A (A = A-B)
  m_sub       macro
              movfp rb+1,WREG ; low byte first
              subwf ra+1,1
              movfp rb,WREG   ; and high byte with borrow
              subwfb ra,1
              endm
main program:
              ; data[j]=(data[i]-h1r)/2
              m_ldai i        ; m_ldai(i)
              m_ldb h1r       ; m_ldb(h1r)
              m_sub           ; m_sub()
              m_div2          ; m_div2()
              m_stai j        ; m_stai (j)
```

WRAP UP

I got the display looking quite good, with an interrupt routine taking ~50% of the available CPU time for video generation. The main program has time to do more than 10 FFT calculations per second, giving a true real-time display.

With my technique, the time spent in the video-generation code depends greatly on the image being displayed. If the screen is full of information, the time left for the main program is near zero. So, you can't use this technique for all video-based projects.

To get satisfactory results with the video-display code, I sacrificed maintainability. Changes to the display may require tremendous efforts in keeping the real-time constraints unchanged. However, it works, and the basic principles may be used to get cheap video display devices like PIC'Spectrum.

For your next video-based project, try to do it with a software video only. If you have strict maintainability requirements, add a CRT controller chip or select a microcontroller with one built in. On the other hand, the FFT implementation is easily reused.

A final note: I'm sure this project wouldn't be possible without a good simulator or an ICE. Thanks to Microchip for providing a good simulator for free on the Web!

Robert Lacoste lives in France, near Paris. He works in the telecommunication industry as senior project manager for GSM wireless base stations development. Robert has 10 years' experience in real-time software and embedded-system development. You may reach him at rlacoste@nortel.com.

SOFTWARE

Complete source code for this article and freeware for the PIC-17C756 in-circuit programmer is available via the Circuit Cellar Web site.

REFERENCES

[1] Microchip Technology, *PIC17C-75X Datasheet*, DS30264A, 1997.

[2] Microchip Technology, *PIC17C-XXX EPROM Memory Programming Specification*, DS30274A, 1997.

[3] S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, New York, NY, 1992.

[4] "VGA Timing Information," www.hut.fi/Misc/Electronics/docs/old/vga_timing.html.

SOURCE

PIC17C756
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com