Peter A. Stark K2OAW
PO Box 209
Mt. Kisco NY 10549

Build a Talking Digital IDer

—K2OAW redesigns his IDer at last

When my CW identifier and repeater control circuit article appeared in the February and March, 1973, issues of *73 Magazine*, I thought that those circuits were about as modern and simple as they could get. Over the years, I've heard of printed circuit boards and kits being sold at hamfests, and several ham repeater manufacturers have used the CW identifier circuit in their systems. The identifier also has been used in RTTY stations to provide Morse code identification.

But times do change; several articles have appeared in *73 Magazine* giving circuits which modified or expanded the original design. I finally decided that it was time for a new identifier design.

Here is an identifier circuit which should renew interest in identifiers for a while. It uses six ICs, the same as the 1973 version, but this identifier talks.

Yes, you read it right. It doesn't whistle or hum your call—it says it right out loud, in plain English, for the whole world to hear. A little muffled, perhaps (after all, what can you expect from six commonly-available ICs?), but clear enough to understand.

I'm having some fun with mine right now. It's sitting on my office desk (with a little IC timer setting it off about once a minute) quietly mumbling "Bah, humbug!" to anyone within earshot!

Although it makes a great conversation piece, that is not its main purpose. I started designing this identifier while driving on a long vacation trip last summer. Every half hour or so, I would remember to key up my 2-meter rig on .52, hoping that somebody would come back. In the meantime, a hundred hams could have passed me by going in the opposite direction. But unless I picked up the mike and gave my call every minute or two, the chances of either one of us knowing about the other were slim. Wouldn't it be nice (I thought) to have an automatic IDer which would key up the rig every minute or so and announce itself? If there were anybody around, they surely would

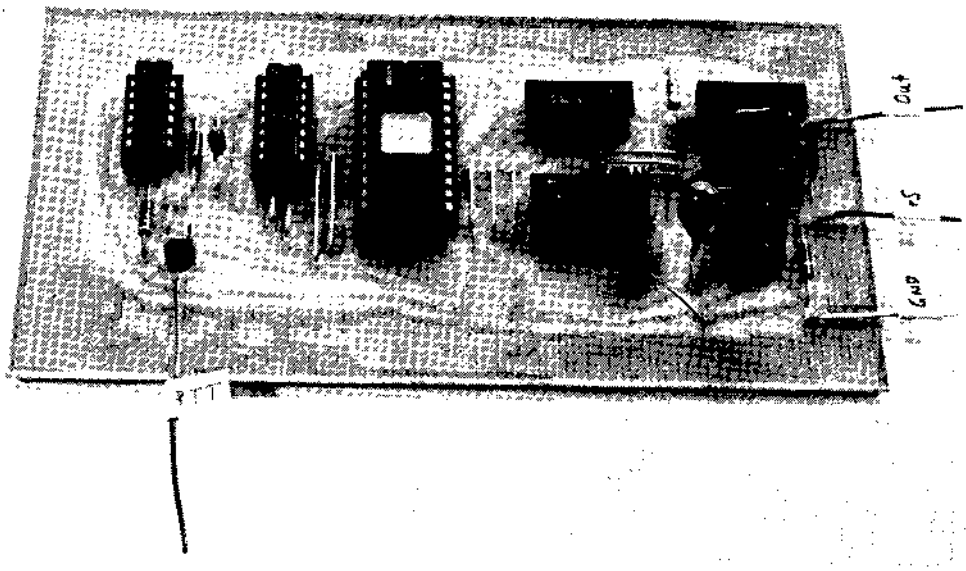

Photo A. Talking identifier.

hear me. And voilà—necessity was the mother of invention.

The identifier uses an EPROM (Erasable Programmable Read Only Memory) to store the voice data to be spoken. The secret, of course, is in knowing how to program this EPROM. I do the programming on my SWTP 6800 computer system, but it could be done on another computer just as well. This article includes the programs and a PC board layout to make your job easier. (Etched and drilled PC boards as well as preprogrammed EPROMs are available from Star-Kits, PO Box 209, Mt. Kisco NY 10549.)

How It Works

There are many ways either to store a real sound recording in a digital memory or to synthesize a fake voice. Quite a few voice synthesizers are available today, ranging from the Texas Instruments Speak and Spell™ to the Computalker synthesizer available for S-100 computers and the Radio Shack synthesizer for the TRS-80. Unfortunately, most of these are fairly complex, require some custom-integrated (and often secret) circuits, and are difficult to program.

Simply storing a digital image of a real voice and playing it back from memory turns out to be much easier and cheaper. That is how this identifier works. Its EPROM contains a digitized "recording" of a voice (which had been digitized previously on a computer), and a fairly simple circuit then scans the memory and "plays" it back. The only problem is to store the voice recording in such a way that it doesn't exceed the capacity of the EPROM.

If memory capacity were not a problem, then the voice pattern could be stored with voice fidelity better than any commercial hi-fi recording. In fact, digital stereo recording is the latest technique on the hi-fi scene because it can provide frequency response and distortion figures beyond anyone's wildest dreams of just a few years ago. But there is a price to be paid—very large amounts of digital data are involved. Digital recording often is done with videotape recorders which can record and play back millions of bits per second. Squeezing two seconds worth of voice into an EPROM which contains just 16,384 bits obviously requires some compromises, and it results in audio quality which is far from hi-fi. But it works.

To see how voice can be digitized, look at Fig. 1(a). Here we see a typical sound waveform such as might be picked up by a microphone. In order to digitize that waveform, we sample it at fixed, periodic intervals, and digitize the voltage that that waveform has at those instants of time.

For instance, suppose we measure the waveform voltage at the points marked with a dot, convert the value of that voltage to a binary number, and store it. If that is later "played" back, we get the waveform shown in Fig. 1(b). The result is a square waveform which changes to a new value at each of the sampling points.

Although the square wave doesn't look anything like the original audio signal, if it is fed through a low-pass filter the sharp corners will be chopped off and the signal will look a bit better.

If, on the other hand, we were to sample the audio signal more often—not only at the dots but also at the intermediate points marked with an X—and digitize that, the resulting waveform, shown in Fig. 1(c), would be a better approximation.

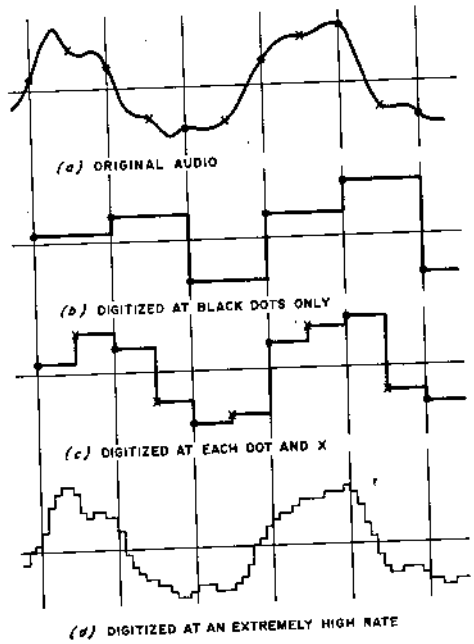

Fig. 1. Digitizing audio at various sampling rates.

Fig. 1(d) shows that when we digitize very often, we get the best waveform yet. Although this waveform does have some sharp corners, they occur at a very high frequency and would be removed very easily with a filter.

How often must we digitize to get an acceptable digitized waveform? There is a rule called the "sampling theorem" which says that the sampling rate must be at least twice the frequency of the highest frequency component in the audio signal. In other words, a hi-fi signal with a frequency response to 20,000 Hz would have to be sampled at least 40,000 times per second. A communications-quality voice signal with a response to 4000 Hz would require sampling at least 8000 times per second.

We can get an idea of this from Fig. 1(b). Sampling at the black dots is enough to get a waveform which follows the large swings of the audio waveform which have a low frequency but cannot capture the small squiggles that have a high-frequency component. To get those, we need a high sampling rate.

Fig. 2 shows a block diagram of the circuitry which would be needed to do the digitizing. Starting with the audio signal, the signal is amplified and sent through a low-pass filter. The purpose of the filter is to remove those frequencies which are too high to be digitized (that is, more than half the frequency of sampling). These components have to be removed to avoid further distortion during the digitizing.

The filtered signal is now sent to a sample-and-hold circuit. This circuit takes a sample of the waveform and holds it in a capacitor while the analog-to-digital (A/D) converter converts the resulting voltage to a binary number. This is necessary because most A/D converters require a steady input voltage while they are converting; if the voltage is changing, then they will probably convert the voltage to the wrong value. Both the sample-and-hold circuit as well as the A/D converter are driven by a

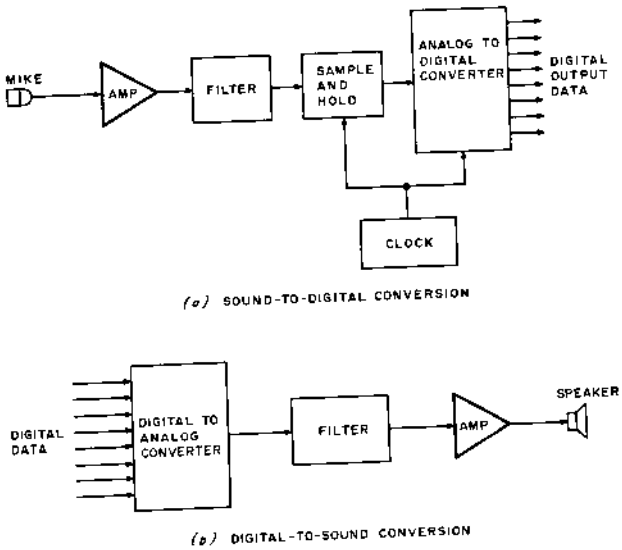

Fig. 2. Circuitry needed to digitize audio.

clock oscillator which sets the rate at which the input signal is sampled.

The output of the A/D converter is now a binary number which can be stored in memory or recorded on tape. When the digitized data is played back, as shown in Fig. 2(b), the binary data is converted back to an analog signal with a digital-to-analog (D/A) converter, passed through a low-pass filter to remove the sharp corners from the wave, amplified, and fed to a speaker.

Now that we know how often a sample should be taken of the input wave, we have another question: How accurately must it be digitized in the A/D converter? This is related to the number of bits produced by the converter for each sample.

A binary number consisting of just one bit can take on only one of two values—either 0 or 1. A binary number consisting of two bits can have values of 00, 01, 10, or 11, a total of four different values. In general, a number which consists of n bits can take on 2^n different values. For instance, ten bits allow 1024 different numbers.

Suppose the converter produces a binary number consisting of just one bit. That one bit is not enough to indicate the precise voltage of the input. With one bit, we can tell only whether the input was positive or negative. This obviously will lead to a very distorted wave, since we cannot hope to keep all the little squiggles in the audio signal.

On the other hand, a ten-bit number can represent 1024 different numbers. Thus, we could measure and encode 512 different positive voltage levels and 512 different negative voltage levels. Thus, the more precise we want our measurements of the sample voltages to be, the more bits we need for each measurement.

In a hi-fi system, we often try to get a signal-to-noise ratio (S/N) of 60 dB or more. 60 dB is a voltage ratio of 1000 to 1, so that we must be able to reproduce two signals even if one is 1000 times larger than the other. This requires being able to measure at least 1000 different positive voltage levels and 1000 different negative voltage levels, for a total of 2000 different voltage levels. Since $2^{11} = 2048$, we need at least 11 bits for this. By the time you add a few more bits to allow these signals to be reproduced with low distortion and to give a little "headroom" so that an occasional burst of extra volume can get through, you are close to 14 bits per sample.

The digital systems being proposed in the hi-fi industry use between 14 and 18 bits per sample; 14 bits are used in consumer products and up to 18 bits are used in the studio-quality recorders which produce the master tapes.

How many bits per second (bps) does this add up to? For pure hi-fi, we need at least 40,000 samples per second, each with at least 14 bits, for a minimum of 560,000 bps (and up to 2 MHz in studio-quality systems). At a rate of 560,000 bps, a 16,384-bit EPROM would provide hi-fi for about 0.03 second. Not enough for a grunt, let alone a ham call.

So we must limit the number of bits per second. This is done by drastically reducing the sampling rate and also reducing the number of bits from the A/D converter.

To squeeze a two-second call into this ROM, we can store 8192 bps. At a sampling rate of 8000 Hz or so (to cover the communications audio range to 4000 Hz), that gives us about one bit per sample. This means that we don't need a complex sample-and-hold circuit, an A/D converter, or even a D/A converter. All we need is some circuit which can tell whether the input audio is positive or negative at the sampling intervals, and which produces a one-bit output—1 if positive, 0 if negative. That turns out to be very simple to do.

The disadvantage is that our voice recording will be very distorted. But by heavily filtering the output with a low-pass filter, we can remove some of that distortion and make the result quite understandable.

The Talking Identifier

Let's leave for a moment the question of how you "record" the voice and store it in the ROM, and look at the circuit of the talking IDer itself, Fig. 3.

The voice pattern is stored in a 2716 EPROM. This is a memory IC currently selling for about $10-$15. It is organized as 2K × 8, meaning that it has 2K storage locations (which is 2048), each holding an 8-bit number.

Each of those 2K locations has an address, a binary number which ranges from 00000000000 to 11111111111. This 11-bit address is fed to the EPROM via the A10 through A0 address pins shown at the bottom of the IC. Each time we give the EPROM an address, it outputs the contents of the addressed location on the eight data lines, D7 through D0, shown on the right side of the EPROM.

The eight bits in the location come out in parallel, meaning all at the same time. But we want the bits one at a time, roughly 1/8000 of a second apart, since each bit represents one sample of the recorded voice pattern. (Over a space of two seconds there is a total of 16,384 samples or bits, which are stored in consecutive locations on the EPROM. The first eight bits are in memory location 00000000000, the next eight bits are in location 00000000001, and so on, up to the last eight bits, which are in location 11111111111.)

The job of splitting up the eight bits in one location into individual bits is handled by the 74LS151 multiplexer. This IC behaves like an SP8T switch which is continuously rotating, scanning the eight bits coming in from the EPROM

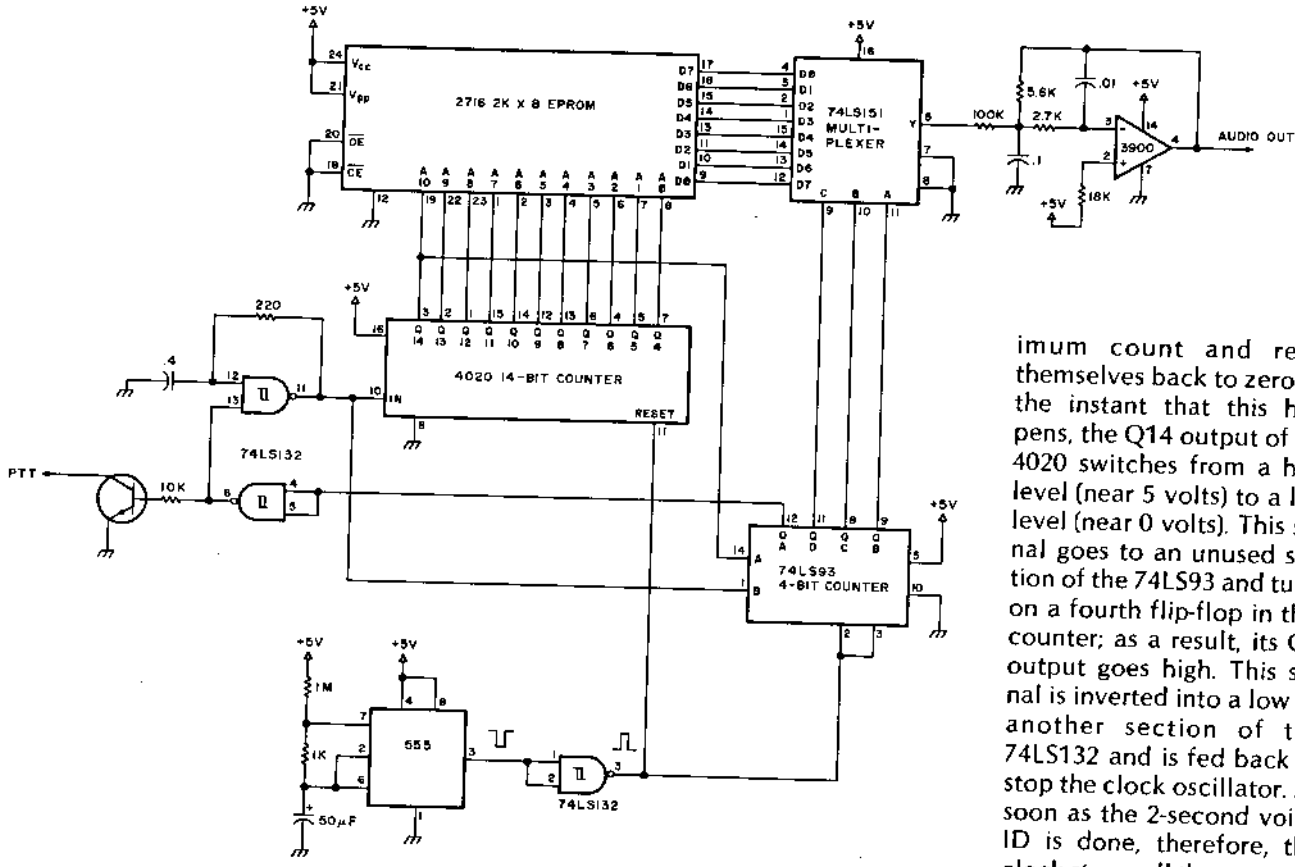

Fig. 3. Talking identifier diagram.

much like the distributor in a V8 car engine. It changes the parallel data coming into the multiplexer into serial data. The result, on pin 5 of the multiplexer, is a square wave which carries the frequency components of the voice but, of course, doesn't have any amplitude information because amplitude was never digitized. This signal is fed into an active low-pass filter which uses an LM3900 Norton op amp, and which cuts off at just under 4000 Hz. This provides the audio output.

The rest of the circuit simply provides different addresses to the EPROM to scan through its memory and also drives the multiplexer.

This part of the circuit starts with one section of a 74LS132 quad, two-input NAND, Schmitt-trigger IC which, along with a 220-Ohm resistor and 0.4-uF capacitor, forms an oscillator which oscillates at about 8 kHz. The output of this oscillator is sent to pin 10 of a 4020 CMOS counter.

The 4020 is a 14-stage ripple counter which contains fourteen flip-flops. Since $2^{14} = 16,384$, this counter can count off 16,384 clock pulses. Since the clock frequency is about 8 kHz, if we start this counter at a count of 0, it will take approximately two seconds to count up to its maximum count. As it does so, it's counting off the 16,384 data bits which are being converted into an audio signal.

We really need 14 outputs from that counter to drive the EPROM address lines and the multiplexer. Unfortunately, to save on pins the 4020 provides only the 11 outputs from the 4th flip-flop (Q4) through the 14th flip-flop (Q14); the outputs of the first three flip-flops are not accessible. So, we have a second counter, which is a 74LS93 binary counter. The oscillator signal which goes to the 4020 goes also to the B input, pin 1, of the 74LS93. Three of the flip-flops in this IC (called B, C, and D) count in parallel with the first three flip-flops of the 4020, and give us the missing signals.

These three signals, on pins 11, 8, and 9 of the 74LS93, change very rapidly and continuously drive the multiplexer which, therefore, scans the output of the EPROM at a high speed (one bit every 1/8000 second).

The eleven bits from the 4020 have a lower frequency and, therefore, drive the address lines of the EPROM at a slower rate (one address every 1/1000 second). Thus, the EPROM feeds out a new group of eight bits every 1/1000 second. Since there are 2K such groups, this again takes about two seconds.

When the two seconds are up, the 4020 and 74LS93 counters reach their maximum count and reset themselves back to zero. At the instant that this happens, the Q14 output of the 4020 switches from a high level (near 5 volts) to a low level (near 0 volts). This signal goes to an unused section of the 74LS93 and turns on a fourth flip-flop in that counter; as a result, its QA output goes high. This signal is inverted into a low by another section of the 74LS132 and is fed back to stop the clock oscillator. As soon as the 2-second voice ID is done, therefore, the clock stops, all the counters (except the A flip-flop in the 74LS93) freeze at zero, and the IDer stops.

The IDer is restarted by resetting all counters to zero with a positive pulse coming out of pin 3 of still another section of the Schmitt trigger NAND. This start signal could be generated externally, but for use with a 2-meter FM rig on 146.52 we have a 555 timer which automatically generates a very short reset pulse every 30 seconds or so. This pulse resets the A flip-flop in the 74LS93, which releases the clock and starts the ID process all over again.

Connected to the clock control line is an NPN transistor. When the clock is running (that is, when the IDer is identifying), that transistor is turned on; when the IDer is off, so is the transistor. By connecting the collector to the push-to-talk (PTT) line of the rig, the IDer automatically keys the transmitter while it is identifying. This circuit is suitable only for

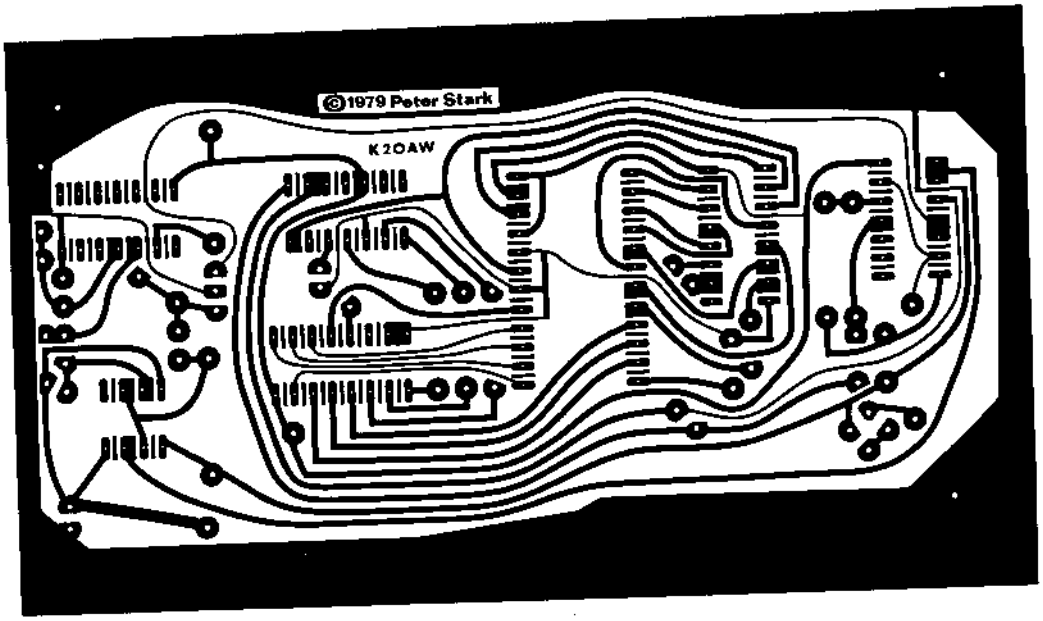

Fig. 4. PC board, copper side.

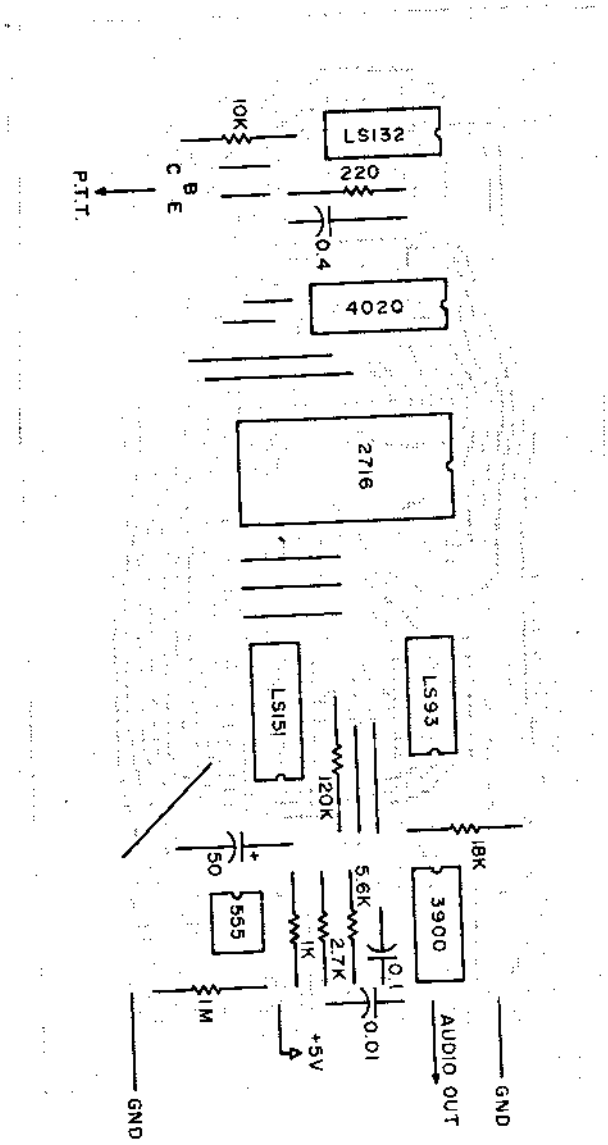

Fig. 5. Parts layout.

driving the PTT line in small, transistorized transceivers. Those rigs which require large currents to drive a PTT relay may require an additional buffer transistor.

Although there are no potentiometers in the circuit, there are several components which may require adjustment. The 100k resistor in the active filter is chosen to provide a fairly small output audio level; if more audio signal is needed, it can be reduced to as low as 5k. Incidentally, do not use disc capacitors in the active filter circuit. Use good quality polystyrene or dipped mica caps.

The oscillation frequency of both the 74LS132 oscillator and the 555 timer depends on the tolerance of the resistors and capacitors used. Since capacitors, especially, tend to have very wide variations, some trimming may be needed to get the right results. To vary the spacing between IDs, you may want to increase or decrease the capacitor value in the 555 timer circuit.

If the 74LS132 oscillator runs too fast or too slow, the voice pattern in the EPROM will be scanned too fast or too slowly, with the same result as when a record is played too fast or too slow. You may like the Donald Duck quality this gives, but for best results you should trim the RC values in this oscillator for the most natural speech sound.

The circuit layout is not critical, and almost any construction method will work, including wire-wrap and temporary prototype socket hookup. If desired, you can use the printed circuit board shown in Fig. 4. Fig. 5 shows the parts layout for the PC board.

The identifier needs approximately 100 mA of +5 volt power. This is provided easily by a three-terminal regulator. If you use the IDer in your mobile, simply include the regulator circuit of Fig. 6. Assuming a load current of 100 mA and a worst-case auto battery voltage of 16 volts, the regulator must drop 11 volts for a power dissipation of 1.1 Watts. With a good heat sink, all this can be dropped in the three-terminal regulator itself; by adding a 39-Ohm, 2-Watt resistor as shown in the circuit, however, we drop 3.9 volts across the resistor. This removes almost .4 Watts of heat from the regulator and dissipates it in the resistor instead.

For applications that require even lower power (such as for battery-powered applications), total circuit power can be reduced even more by lifting the chip enable pin (pin 18) of the 2716 from ground and connecting it instead to pin 12 of the 74LS93. This disables the 2716 when the circuit is not identifying. The circuit still draws around 100 mA when identification is in progress, but cuts it down to less than half during other times.

"Recording" the EPROM

To digitize the audio signal, we need a filter to remove high-frequency components above 4000 Hz and a comparator circuit to

sense the polarity of the input audio. This circuit uses another LM3900 quad Norton op amp and is shown in Fig. 7.

One op-amp in the LM3900 is used as an active low-pass filter with a cutoff frequency of just under 4000 Hz. This amplifier/filter has a small amount of gain but not enough to accept the weak signal from a microphone. It is designed for use with an external mike preamp or with the higher-level output of a tape recorder. I generally record the desired message on tape first and then feed the speaker output of the recorder to the audio input of this circuit.

A battery-operated recorder is best in this case, since with a high gain it is possible for hum to be digitized between words. Hum gets swamped out during speech, but when there is silence, the circuit works much like a volume compressor by boosting low-level sounds. Thus, a good S/N ratio is essential. The 10k volume control on the input helps to cut down excessive signal; its correct adjustment is important.

The output of the filter is sent to another op-amp section of the LM3900, which is used as a slicer or comparator. The signal coming from the filter is sent to one input of this op amp while a reference current from the 10k zero-set pot is fed to the other. As the filtered audio output goes above or below the reference signal, the digital output from pin 9 switches between 0 and +5 volts.

The 10k zero-set pot should be adjusted so that with the audio input shorted to ground, the output is just on the verge of switching between 0 and +5 volts. With proper adjustment, positive audio peaks will clip the digital output one way while negative peaks flip it the other way.

For testing purposes, an audio amplifier/speaker combination can be connected to the digital output to monitor the signal after it has been digitized; I use an inexpensive Radio Shack signal tracer for this purpose. The digitized signal is supposed to be filtered before being heard, so this signal will sound excessively harsh, but it is good enough to give you an idea of whether the circuit is working.

Once we have the one-bit digital output, we must sample it at intervals of about 1/8000 second, convert the samples into 8-bit bytes, and store them. Before burning them into the EPROM, however, it is very convenient to be able to "play" them back to make sure that the volume controls have been set right and that we have the right voice segment. It also would be very convenient if in some way we could edit the digital code to eliminate any noise just before and after the call. In other words, it would be very convenient if we could store the message in RAM and read or modify it before it is permanently stored in EPROM.

Building a special piece of hardware for just this purpose is difficult and expensive. Fortunately, most home or personal computers have an input and output port which could be used to input or output this one-bit digital signal and also have RAM which could be used to store the code temporarily. This makes the job almost trivial.

To do this, you need a program which will input data, group bits together in sets of 8, and store them. In most cases, this program has to be written in machine or assembly language since most BASIC systems are not fast enough to take 8000 samples per second and process them.

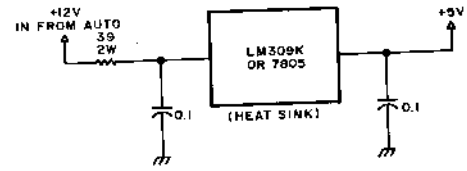

Fig. 6. Voltage regulator for mobile use.

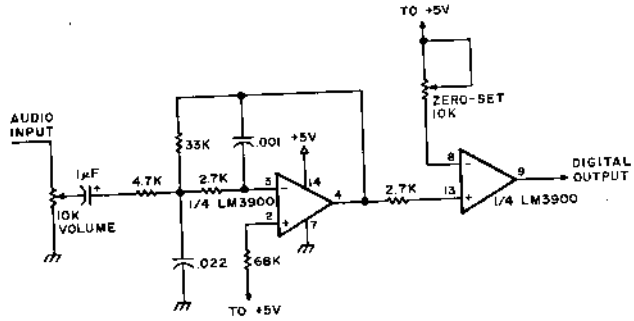

Fig. 7. Audio-to-digital conversion circuit.

Obviously, the program will depend on the particular computer used, but as a starter, I'm including here three programs written for an SWTP 6800 system which are very useful.

Parallel input/output on 6800 systems is usually handled by an IC called a PIA or Peripheral Interface Adapter. Although this IC has twenty input/output pins, only two are used in this application—bit 0 of port A gets the input from the circuit of Fig. 7, while bit 1 of the same port feeds an audio amplifier/speaker combination which is used to listen to the recorded sound.

Program 1 is an echo program which is used only for testing. It inputs via bit 0, outputs the bit right back to bit 1 of the input/output port, and then waits for a short while to simulate the delay between samples. When everything is running correctly, the audio coming out of the computer will sound very similar to the audio you could hear directly at the output of the circuit of Fig. 7. (It, too, will sound harsh because of the lack of filtering.)

The program starts by initializing the PIA to set up the correct bits for input and output. The main part of the program (starting at

```
                ********************************************
                *                                          *
                * THIS PROGRAM INPUTS DATA FROM PORT A BIT 0 *
                * OF A PIA IN PORT 7, AND ECHOES IT TO BIT 1 *
                *                                          *
                ********************************************

(B01C)    PIADAT EQU   $B01C     PORT A DATA/DIRECTION REG
(B01D)    PIACTL EQU   PIADAT+1  PORT A CONTROL REGISTER

(0100)           ORG   $0100
0100 7F B01D START CLR   PIACTL  RESET PIA
0103 86 02       LDA A #$2      SET BIT 0=INPUT, BIT 1=OUTPUT
0105 B7 B01C     STA A PIADAT
0108 86 04       LDA A #$4      RESET BACK TO DATA
010A B7 B01D     STA A PIACTL
010D B6 B01C LOOP LDA A PIADAT  LOAD DATA FROM PORT A BIT 0
0110 48          ASL A          SHIFT LEFT INTO BIT 1
0111 B7 B01C     STA A PIADAT   OUTPUT TO PORT A BIT 1
0114 8D 02       BSR   WAIT
0116 20 F5       BRA   LOOP     GO BACK AND REPEAT

                * FOLLOWING WAIT ROUTINE INTRODUCES A DELAY
                * WHICH PERMITS SAMPLING RATE TO BE CHANGED

0118 CE 0010 WAIT  LDX   #$0010 INITIALIZE INDEX REGISTER
011B 09      WAIT1 DEX        DECREMENT INDEX
011C 26 FD       BNE   WAIT1   REPEAT IF NOT YET ZERO
011E 39          RTS           OTHERWISE RETURN
```

Program 1. Echo test.

```
                ********************************************
                *                                          *
                * THIS PROGRAM INPUTS DATA FROM PORT A BIT 0 *
                * OF A PIA IN PORT 7, PACKS 8 BITS PER BYTE, *
                * AND STORES THE DATA IN MEMORY LOCATIONS 1000 *
                * TO 7FFF.                                 *
                *                                          *
                ********************************************

  (801C)    PIADAT EQU   $801C     PORT A DATA/DIRECTION REG
  (801D)    PIACTL EQU   PIADAT+1  PORT A CONTROL REGISTER

  (0100)           ORG   $0100
0100 7F 801D START CLR   PIACTL    RESET PIA
0103 86 02         LDA A #$2      SET BIT 0=INPUT, BIT 1=OUTPUT
0105 B7 801C       STA A PIADAT
0108 86 04         LDA A #$4      RESET BACK TO DATA
010A B7 801D       STA A PIACTL
010D CE 1000       LDX   #$1000   POINT TO MEMORY BUFFER ADDRESS
0110 C6 08   LOOP1 LDA B #$08
0112 F7 0132       STA B BITCTR   COUNT 8 BITS PER BYTE
0115 4F            CLR A          ERASE A ACCUMULATOR
0116 F6 801C LOOP2 LDA B PIADAT   READ DATA INTO B ACCUMULATOR
0119 C4 01         AND B #$01     MASK OFF EVERYTHING EXCEPT BIT 0
011B 48            ASL A          SHIFT A ACCUM LEFT
011C 1B            ABA            ADD NEW BIT FROM B TO A
011D C6 10         LDA B #$10     SET UP COUNTER FOR SAMPLING DELAY
011F 5A      WAIT  DEC B          DECREMENT B
0120 26 FD         BNE   WAIT     REPEAT IF NOT YET ZERO
0122 7A 0132       DEC   BITCTR   DO FOR 8 BITS
0125 26 EF         BNE   LOOP2    GET NEXT BIT
0127 A7 00         STA A 0,X      STORE BYTE WHEN COMPLETED
0129 08            INX            INCREMENT INDEX REGISTER POINTER
012A 8C 7FFF       CPX   #$7FFF   CHECK FOR END OF MEMORY
012D 26 E1         BNE   LOOP1    REPEAT IF OK
012F 7E E0D0       JMP   $E0D0    RETURN TO MONITOR WHEN DONE

0132          BITCTR RMB   1      BIT COUNTER TO COUNT 8 BITS
```

Program 2. Input.

```
                ********************************************
                *                                          *
                * THIS PROGRAM GETS DATA FROM MEMORY       *
                * LOCATIONS 1000-7FFF, UNPACKS IT INTO     *
                * INDIVIDUAL BITS, AND OUTPUTS TO PORT A   *
                * BIT 1 OF A PIA IN PORT 7.                *
                *                                          *
                ********************************************

  (801C)    PIADAT EQU   $801C     PORT A DATA/DIR REGISTER
  (8019)    PIACTL EQU   PIADAT+1  PORT A CONTROL REGISTER

  (0180)           ORG   $0180
0180 7F 801D START CLR   PIACTL    RESET PIA
0183 86 02         LDA A #$2      SET BIT 0=INPUT, BIT 1=OUTPUT
0185 B7 801C       STA A PIADAT
0188 86 04         LDA A #$4      RESET BACK TO DATA
018A B7 801D       STA A PIACTL
018D CE 1000       LDX   #$1000   POINT TO MEMORY BUFFER ADDRESS
0190 C6 08   LOOP1 LDA B #$0B
0192 F7 01B4       STA B BITCTR   COUNT 8 BITS PER BYTE
0195 A6 00         LDA A 0,X      GET NEXT BYTE FROM MEMORY
0197 16      LOOP2 TAB            TRANSFER IT TO B REGISTER
0198 48            ASL A          SHIFT A ACCUM LEFT 1 BIT
0199 59            ROL B          ROTATE B LEFT 3 BITS TO MOVE THE CURRENT
019A 59            ROL B          BIT FROM BIT 7 (LEFT-MOST) INTO
019B 59            ROL B          BIT 1 (SECOND FROM RIGHT)
019C C4 02         AND B #$02     MASK OFF EVERYTHING EXCEPT BIT 1
019E F7 801C       STA B PIADAT   OUTPUT TO PIA
01A1 C6 08         LDA B #$0B     SET UP COUNTER FOR SAMPLING DELAY
01A3 5A      WAIT  DEC B          DECREMENT B
01A4 26 FD         BNE   WAIT     REPEAT IF NOT YET ZERO
01A6 7A 01B4       DEC   BITCTR   DO FOR 8 BITS
01A9 26 EC         BNE   LOOP2    IF BIT COUNTER NOT ZERO
01AB 08            INX            INCREMENT INDEX WHEN BYTE IS DONE
01AC 8C 7FFF       CPX   #$7FFF   CHECK FOR END OF MEMORY
01AF 26 DF         BNE   LOOP1    REPEAT IF OK
01B1 7E E0D0       JMP   $E0D0    RETURN TO MONITOR WHEN DONE

01B4          BITCTR RMB   1      BIT COUNTER TO COUNT 8 BITS
```

Program 3. Output.

the statement labeled LOOP) loads a bit from the PIA, shifts it left from bit 0 into bit 1, and outputs it. Then it branches to a WAIT subroutine for a short delay, after which it branches back to LOOP.

For experimental purposes, it's important to be able to calculate how many samples are taken per second. This is done by computing how many computer clock cycles are required for each instruction in the loop. In Program 1, the main loop takes 31 clock cycles plus 8 cycles for each repetition of the WAIT1 loop. With the WAIT1 loop initialized (with the LDX instruction) to run 16 times (0010 hexadecimal), the total time between samples is 31 + (16) × (8) = 159 clock cycles.

In a typical SWTP computer running with a 900-kHz clock, each clock cycle takes 1.11 microseconds, so that the total delay between samples is 177 microseconds; this translates into a sampling rate of about 5600 samples per second, which is about the minimum that can be used for acceptable results. For 8000 samples per second, the LDX instruction should be changed to run the WAIT1 loop 10 times.

Once the echo test program reveals that the A/D conversion and the computer input/output circuitry is working correctly, Program 2 can be used to input data into the computer's memory, while Program 3 is used to output it back to the speaker. Both of these programs have a WAIT loop which provides some control over the delay between samples. There is some leeway here in adjusting this delay. If the number of samples taken per second is changed above or below 8000 (to increase playing time, for instance), the clock oscillator frequency in the identifier circuit of Fig. 3 also has to be changed to a similar value or the final output will have a pitch which is too high (like Donald Duck) or too low.

Both programs are located in low memory, with the input program starting at location 0100 (hex) and the output program at 0180. They do not overlap and, therefore, can be in memory at the same time. Thus, we can input audio, store it in memory, and then output it right back.

The programs are written for a 32K computer and use locations 1000 (hex) through 7FFF to store the resultant digital data. This is a total of 28K of memory; at the rate of 1K per second, this can store a total of 28 seconds of sound. When Program 2 is finished, it returns to the monitor. Rather than calculate the sampling rate by computing the number of cycles per loop, etc., an easier way to adjust the WAIT loop is to note how long the overall program runs. If it runs exactly 1 second per K of memory used, then it is running at 8192 samples per second.

By changing the starting address (1000 hex) or the ending address (7FFF) in Program 3, we can "play" back just selected portions of the input. In this way, we can pick one of several versions of the same call, choosing the one that sounds best. This allows us to edit the data before it is stored into EPROM. Once you find the portion which sounds best, burn that portion into the EPROM and keep the rest of the EPROM data empty (an erased 2716 EPROM has a hex FF in every location). This will assure that no noise or sounds are in the EPROM other than the actual call.

Conclusions

While this talking identifier won't win any awards for hi-fi quality, it is perfectly understandable and fulfills its purpose well. It also gives you a chance to experiment with speech reproduction via digital means. In addition, it's a lot more satisfying to build such a device from commonly-available ICs than to go out and buy an expensive synthesizer chip or system. Why don't you try it?

So, if you ever hear something grumble "K2OAW" on 146.52 as I speed by your house on the nearby Interstate, maybe you'll be able to turn on your own IDer and have it come back to me. ■