C=Hacking #20

```
                  ########
              ##################
         ######            ######
       #####
    #####  ####  ####      ##      #####   ####  ####  ####  ####  ####
#####
  #####    ##    ##      ####    ##   ##   ##  ###     ##    ####  ##   ##
##
 #####    ########     ##  ##   ##        #####       ##    ## ## ##   ##
#####    ##    ##    ########  ##   ##   ##  ###     ##    ##  ####   ##
##
#####  ####  ####  ####  ####  #####   ####  ####  ####  ####  ####   ######
#####                                                                   ##
 ######            ######           Issue #20
   ##################             April 18, 2001
       ########
```

.....
...

Ineptitude: If you can't learn to do something well, learn to enjoy
doing it poorly.
-- "Demotivator" poster

.....
...

BSOUT

Hoo-ah! Time for another late (but hopefully great) issue of C=Hacking!

This issue features several nifty articles on both software and hardware projects. There seems to be a lot of activity in the hardware area right now, so hopefully we'll see more hardware articles in future issues.

In the software area, however...

If I may pithily berate for a moment, I'd like to observe that talking about programming on comp.sys.cbm is -- this may come as a surprise to some -- not the same as actually programming. I'd like to once again encourage those who
have been talking about projects, or have half-completed projects sitting around on an FD disk somewhere, to go for it and get the job done!

Just like... um... C=Hacking (hey, it gets done... eventually...).

```
The format for this issue has changed a little, with all the news and stuff
moved into the previously skimpy "Jiffies" section.  Therefore, this is now
just the 'me' column.

So 'me' would like to thank all the authors in this issue for their time
and effort (and patience), and all of the true C= Hackers out there for
their
spirit and work and cool projects.

'Me' is also very happy to announce that he is getting married on August 17.
And heck, you're ALL INVITED (I think an SX-64 would make an excellent
wedding present, don't you?).

And finally, me still thinks the 64 is the niftiest computer ever made.

Onwards!

-Steve

.......
....
..
.                                     C=H #20
```

Contents

```
BSOUT
    o Voluminous ruminations from your unfettered editor.


Jiffies
    o News, things, and stuff.


Side Hacking

    o "Super/Turbo CPU VDC Hack", by Henry Sopko <henry.sopko@hwcn.org>
      Normally it is not possible to access the VDC chip when using
      a SuperCPU64 or Turbo CPU on a 128.  The 1-wire hack described
      in this article fixes that situation!

    o "16K Memory Expansion for the VIC-20", by Per Olofsson
      <MagerValp@cling.gu.se>.  This article describes a nifty way to
      add more memory to the VIC-20, along with some basic circuit
      design information for the hardware neophyte (i.e. people like me!).

    o "Quick Quadratic Splines", by moi <sjudd@ffd2.com>
      A spline is a powerful tool for drawing a curve through an arbitrary
      set of points -- for motion/animation, for arbitrary curves (like
```

```
      fonts), and numerous other tasks.  This article describes _quadratic_
      splines and some fast C64 implementations, and includes a program
      for experimenting with splines.


Main Articles

    o "VIC-20 Kernal ROM Disassembly Project", by Richard Cini
      <rcini@email.msn.com>

      The ever-dependable Richard Cini has written the fourth article
      in the quest for a complete disassembly of the VIC-20 kernal.  This
      installment focuses on device I/O routines: SETNAM, SETLF, OPEN,
      and beyond.

    o "MODs and Digital Mixing", by Jolse Maginnis
                        <jmaginni@postoffice.utas.edu.au>.

      Josmod is a program for JOS, by Jolse, that can play Amiga MOD files
      (and their newer successors).  This article describes the general
      functioning of the program, the layout of a MOD file, and how to mix
      multiple digital samples in real-time (and hence play MODs!).

    o "The C64 Digi", by Robin Harbron <macbeth@psw.ca>, Levente
      Harsfalvi <levente@terrasoft.hu>, and Stephen Judd <sjudd@ffd2.com>

      This article is, we hope, a complete reference on digital sampling
      and the C64, including: general theory, SID hardware description,
      and methods of playback (changing $d418, pulse width modulation,
      and various tricks).  Numerous code examples are given, along with
      a program that does true 8-bit playback at 16KHz -- it requires a
      SuperCPU, but it is most impressive, and chances are awfully good
      that you've never heard a digi like this out of SID before.
```

Credits

```
Editor, The Big Kahuna, The Car'a'carn..... Stephen L. Judd
C=Hacking logo by.......................... Mark Lawrence

Special thanks to the folks who have helped out with reviewing and such,
to the article authors for being patient, and to all the cbm hackers that
make this possible!

Legal disclaimer:
    1) If you screw it up it's your own fault!
    2) If you use someone's stuff without permission you're a dork!

About the authors:
```

Jolse Maginnis is a 20 year old programmer and web page designer, currently taking a break from CS studies. He first came into contact with the C64 at just five or six years of age, when his parents brought home their "work" computer. He started out playing games, then moved on to BASIC, and then on to ML. He always wanted to be a demo coder, and in 1994 met up with a coder at a user's group meeting, and has since worked on a variety of projects from NTSC fixing to writing demo pages and intros and even a music collection. JOS is taking up all his C64 time and he is otherwise playing/watching sports, out with his girlfriend, or at a movie or concert somewhere. He'd just like to say that "everyone MUST buy a SuperCPU, it's the way of the future" and that if he can afford one, anyone can!

Richard Cini is a 31 year old vice president of Congress Financial Corporation, and first became involved with Commodore 8-bits in 1981, when his parents bought him a VIC-20 as a birthday present. Mostly he used it for general BASIC programming, with some ML later on, for projects such as controlling the lawn sprinkler system, and for a text-to-speech synthesyzer. All his CBM stuff is packed up right now, along with his other "classic" computers, including a PDP11/34 and a KIM-1. In addition to collecting old computers Richard enjoys gardening, golf, and recently has gotten interested in robotics. As to the C= community, he feels that it is unique in being fiercely loyal without being evangelical, unlike some other communities, while being extremely creative in making the best use out of the 64.

Robin Harbron is a 28 year old internet tech support at a local independent phone company. He first got involved with C= 8-bits in 1980, playing with school PETs, and in 1983 his parents convinced him to spend the extra money on a C64 instead of getting a VIC-20. Like most of us he played a lot of games, typed in games out of magazines, and tried to write his own games. Now he writes demos, dabbles with Internet stuff, writes C= magazine articles, and, yes, plays games. He is currently working on a few demos and a few games, as well as the "in-progress-but-sometimes-stalled-for-a-real-long-time-until-inspiration-hits-again Internet stuff". He is also working on raising a family, and enjoys music (particularly playing bass and guitar), church, ice hockey and cricket, and classic video games.

Levente Harsfalvi is a 26 year old microcontroller programmer who works at a small local company. His first C= encounter was a Plus/4 at school, at the age of 12, and later (1988) his parents bought a C-16. After learning BASIC and ASM coding he joined a Plus/4 demo group (GOTU, and later Coroners),
and has worked on game conversions, an FLI editor, music software (including a SID emulator to play c64 music on TED) and numerous other software and hardware projects. More recently he has begun taking some measurements on the Plus/4 to figure out things such as how the sound generator works and is working on a C-16 demo. Outside of the C= he enjoys cycling and running, and ~50km walking tours.

```
For information on the mailing list, ftp and web sites, send some email
to chacking-info@jbrain.com.

While http://www.ffd2.com/fridge/chacking is the main C=Hacking homepage,
C=Hacking is available many other places including

    http://www.funet.fi/pub/cbm/magazines/c=hacking/
    http://metalab.unc.edu/pub/micro/commodore/magazines/c=hacking/
```

Jiffies

```
$01 Not _too_ long ago, an effort was made to write down the pin assignments
    for the video port of all the major C= computers.  The result, as
compiled
    by William Levak <wlevak@cyberspace.org>, is:


            8              7
Commodore           6           Video Connector
            3              1

              5         4
                   2
                                                    Plus4     C16/C128
     CBM-II      VIC20     VIC20CR    C64     Pin C64A/SX64   C64B/C/E
    --------- ------------ ------- ---------- --- ---------- ----------
(R) Luminance +5 V         +5 V    Monochrome  1  Monochrome Monochrome (Y)
    Ground    Ground       Ground  Ground      2  Ground     Ground
(B) V. Sync.  Audio        Audio   Audio       3  Audio      Audio      (W)
(W) Video     50 Ohm Video Video   Video       4  Video      Video
(Y) H. Sync.  Video        Video   Audio In    5  Audio In   Audio In
                                               6  Chroma     Chroma     (R)
                                               7
                                               8             +5 V


$02 "Professor Dredd" <profdredd@yahoo.com> has uploaded the programs
    from "Inside Commodore DOS" to his webpage at

    http://www.geocities.com/profdredd


$03 Todd Elliot has written a version of Pasi Ojala's zip/unzip program
    for GEOS/Wheels, available at:

    http://www.cs.tut.fi/~albert/Dev/gunzip-geos/


$04 Jeri has been hard at work on her video/cpu-board:
```

```
    http://www.geocities.com/cm_easy/


$05 Jolse has been hard at work on JOS (but you'll have to wait until next
    issue for the article!):

    Here's the latest Jos news:
    Support for CMD HD, including native partitions. (Read only atm, and no
    1581 partitions)
    Enhanced shell with filename completion and recursive wildcards.
    Improvements to the GUI - the architecture now allows for a different
    window interface styles transparent to the application.
    Numerous showstopper bugs killed.
    Improvements to the httpd server to allow directory listings.
    Swiftlink/T232/Duart drivers.
    Started writing tutorials for programmers wanting to give Jos a go.

    Plus heaps more things not worth mentioning..

    cya!,
    Jolse


$06 Philip 'Pepto' Timmermann has made some very accurate measurements
    (and RGB calculations) of VIC-II colors:

    http://www.pepto.de/projects/colorvic/


$07 And finally, I have written a 2D graphics library for use by assembly
    language programs (plot points, draw lines and circles, that kind of
    thing).  It's super-easy to use and pretty fast, so go ahead and use it
    if you need some hires drawing routines!

    For more information, pop on over to

    http://www.ffd2.com/fridge/grlib/
```

Side Hacking

Super/Turbo CPU VDC Hack

```
                        Super/Turbo CPU VDC Hack
                                 for
        Commodore 128 with SuperCPU 64 or Turbo Master CPU Accelerators
```

by Henry Sopko
(henry.sopko@hwcn.org)

As many of you probably know, accessing the VDC (8563) chip is not possible when using a SuperCPU 64 (version 1 tested only!) or the TurboMaster CPU (latest revision), until now. With this 1 wire hack, you can have access to the C128 (flat only tested) VDC 80 column chip with your SCPU 64 or TurboMaster CPU!

DISCLAIMER

I take no responsibility whatsoever to any damage that may occur to your Computer or SCPU 64/TM CPU resulting from this hardware hack! You do this hack totally at YOUR OWN RISK!

[C=Hacking disclaimer: if you screw it up, it's your fault!]

This hack was only tested on a flat C128, using CMD's SCPU 64 (version 1). Tested also was the Turbo Master CPU 4.09 MHz accelerator from Schnedler Systems (latest revision).

Parts required: (1) long wire (12 inches or so, cutting it to length).

INSTRUCTIONS

Dissassemble your C128, taking the metal shield completely off. With the motherboard exposed, find the 8563 VDC chip (U22). After locating this chip, remove it (take note of the notch position, so you correctly re-insert the chip). Bend out PIN 9 (R/W) of the 8563 just enough so it does not touch the socket or any metal (in the flat 128, theres not much room, so be carefull).

Now re-insert the 8563 chip. Take a wire (you can either solder the wire to pin 9 as I did, or use Microclips -- your choice). Now, connect the wire by soldering or using Microclips to PIN 5 on the CARTRIDGE EXPANSION PORT. Thats
it!

I have done this hack quite a while ago without any signs of problems whatsoever. The C128 functions the same with or without a SCPU 64 or TM CPU connected after preforming this hack. I really did this hack for the Turbo Master CPU so I could access the VDC. Turns out, that the SCPU 64 accelerator
(and maybe others?) work with this hack as well. The better choice of course is to buy a CMD SuperCPU 128 to take full advantage of the Commodore 128 in both modes!

16K Memory Expansion for the Vic-20

By Per Olofsson <MagerValp@cling.gu.se>.

Thanks to Ruud Baltissen, Pasi Ojala and Spiro Trikaliotis for help and suggestions. The latest version of this project may be found at http://www.cling.gu.se/~cl3polof/64.html.

I tried to keep the expansion as simple as possible, requiring no chips besides the two memory chips, and a minimum of soldering. It uses two 6264 SRAM chips (62256 also works) piggy-backed to the Kernal and Basic 2364 ROMs. It's recommended that you do the expansion one chip at a time, as this greatly simplifies troubleshooting. After the first chip checks out OK, do the
second one. I'll start out by explaining a few basic principles.
o Static Electricity

We'll be using CMOS RAM chips, and they are very sensitive to static electricity. If you don't have an anti-static wrist strap, make sure you touch
a grounded surface before you touch the chips. Do not work sitting on shag carpet in a mohair sweater while petting the cat :)
o Be Careful With ICs

ICs are sensitive to heat, and keeping the soldering iron for too long on a pin can toast the chip. If you need to redo a solder joint on a pin, wait for
it to cool down first.
o Up, Down, and Pin Numbering

The top of a DIP IC (the kind used in Vic-20s) is marked with a small notch, looking something like this:

```
     ___ ___
  1 =|   V   |=  8
  2 =|       |=  7
  3 =|       |=  6
  4 =|       |=  5
     `-------'
```

As you can see, the pin numbers start at 1, and the first pin is the top left,
going down to the bottom left, then from the bottom right to the top right.

Piggy-backing

Piggy-backing is a quick way of adding another IC on top of an existing one. Normally you would create a PCB with chip sockets, connect it to the expansion port and populate it with memory chips, but with piggy-backing you

simply add chips on top of internal ones. This works when the new chip's pins
has signals that are identical to, or closely matches, the layout of the one it's being mounted on top of.

```
                                                        6264 RAM
                                                       ___ ___
            2364 ROM                          NC   1  =|   V   |=  28  Vcc
                ___ ___                      A12   2  =|       |=  27  /WE
   A7   1  =|   V   |=  24  Vcc              A7   3  =|       |=  26  CS2
   A6   2  =|       |=  23  A8               A6   4  =|       |=  25  A8
   A5   3  =|       |=  22  A9               A5   5  =|       |=  24  A9
   A4   4  =|       |=  21  A12              A4   6  =|       |=  23  A11
   A3   5  =|       |=  20  /CS              A3   7  =|       |=  22  /OE
   A2   6  =|       |=  19  A10              A2   8  =|       |=  21  A10
   A1   7  =|       |=  18  A11              A1   9  =|       |=  20  /CS1
   A0   8  =|       |=  17  D7               A0  10  =|       |=  19  D7
   D0   9  =|       |=  16  D6               D0  11  =|       |=  18  D6
   D1  10  =|       |=  15  D5               D1  12  =|       |=  17  D5
   D2  11  =|       |=  14  D4               D2  13  =|       |=  16  D4
  Vss  12  =|       |=  13  D3              Vss  14  =|       |=  15  D3
            `-------'                                 `-------'
```

As you can see there are 8 that don't match. However, we don't have to rewire
all of them. NC means "No Connection" so we can just ignore that pin. Note that if you're using 62256 chips you'll have to wire this one too, see below. We want to connect CS2 to Vcc, and we'll also swap A11 and A12, leaving
5 pins to solder on each chip. Swapping address bus pins works on RAM chips, as it only affects how bits are stored internally -- when you read them out again, they're swapped back. This also works for the databus.
We'll mount one 6264 on top of the Kernal ROM, and one 6264 on top of the Basic ROM. The ROMs are marked UE11 and UE12 and can be found in the bottom right of the motherboard. The wiring is identical for the two chips, except for the /OE and /CS1 pins. The pins that we want to rewire we carefully bend up so that they don't connect to the ROM. You want to bend them almost all the
way up (about 150 degrees) so that you can reach them with the soldering iron. The pins are very sensitive, so make sure you bend the right pins -- bending a pin back again could easily break it.
Sometimes the pins on the RAM are too wide apart to make a good connection when you piggy-back it. In this case, bend all the pins on the RAM slightly inwards. You can do this by putting it on the side on a flat, hard surface and
press gently.
o Soldering A11/A12 and Vcc

You could get A11, A12, and Vcc from several places on the motherboard, but as
they're available on the ROMs we'll just solder small wires from the ROM to

the RAM. Remember that we're swapping A11 and A12, so connect pin 18 on the ROM to pin 2 on the RAM. Connect pins 26 and 28 on the RAM to eachother.
o Soldering /WE

Pin 27 on the RAMs should be connected to pin 34 on the 6502 CPU. The CPU is the 40-pin chip in socket UE10 on the motherboard, right next to the ROM chips. Pin 34 is the 7th pin if you count from the top right pin on the CPU.
o /OE and /CS1

In the Vic-20 memory is divided into 8 blocks of 8K each. Block 0 is further divided into 8 1K blocks, of which 5 are populated with RAM. Block 4 is used
by the VIC and the VIA chips, block 6 is the Basic ROM, and block 7 is the kernal ROM. This leaves four blocks (1, 2, 3 and 5) available for cartridges and RAM expansions. For RAM to be visible to the basic interpreter, you must start by adding ram in block 1, then 2 and then 3. RAM in block 5 is never detected by the basic. 8K cartridges use block 5, and 16K cartridges use block
5 together with another one, usually 3. To be as compatible as possible with existing cartridges and to expand basic memory I'll use block 1 and 2 for this
expansion, but you could use any two of the available blocks you want. I've added instructions for making the blocks selectable by switches below.
The block signals are available on the 74LS138 decoder chip marked UC5 on the
left side of the motherboard. Block 1 is on pin 14, block 2 is on pin 13, block 3 is on pin 12 and block 5 is on pin 10. If you look closely you'll see
that the signals for block 1 and 2 go out from the chip a few mm to a small pad. It's much easier to connect your wires to these pads than the pins on the
decoder chip. Solder a wire from block 1 to pin 20 and 22 on the first RAM chip, and a wire from block 2 to pin 20 and 22 on the second RAM chip.
You're done!

That's all. When you power up the Vic-20, you should be greeted with a 19967 BYTES FREE message. If you've only done one chip so far, you'll get 11775 BYTES FREE, provided you connected it to block 1. If you've connected memory to other blocks but not block 1, you'll just get the normal 3583 BYTES FREE message. To test your memory, try this program:

```
  10 input "test which block";b
  20 s = b*8192 : t = s+8191 : e = 0
  30 print "testing databus"
  40 for a = s to t : poke a, 85 : if peek(a)  85 then gosub 1000
  50 poke a, 170 : if peek(a)  170 then gosub 1000
  60 next : de = e
  70 e=0 : print "testing high address bus"
  80 for a = s to t : poke a, int(a/256) : next
  90 for a = s to t : if peek(a)  int(a/256) then gosub 1000
  100 next : he = e
```

```
  110 e=0 : print "testing low address bus"
  120 for a = s to t : poke a, a and 255 : next
  130 for a = s to t : if peek(a)  a and 255 then gosub 1000
  140 next : print
  150 print de;"errors found in databus test"
  160 print he;"errors found in high address bus test"
  170 print e;"errors found in low address bus test"
  999 end
  1000 e = e+1 : print "error at";a : return

The program takes a couple of minutes to run.
Troubleshooting
---------------

  The computer doesn't start at all, or all you get is a black screen
   You've probably shorted or toasted something. Not good, this could
   have damaged the computer. Recheck all your soldering and make sure
   that you haven't accidentally connected something wrong.
  The computer powers up with 3583 bytes free
   Memory in block 1 is either not connected or not working. Check the
   connections between block 1, /CS1 and /OE, between R/W and /WE, and
   between Vcc, CS2 and Vcc.
  The computer powers up with something other than 3583, 11775 or 19967
bytes
  free
   Memory is functioning partially, check A11 and A12. This could also
   indicate that a RAM chip is faulty.
  The computer powers up with the correct number of bytes free, but the
memory
  test program fails
   Memory is functioning partially. If the databus test fails, check the
   connections between block 1, /CS1 and /OE, between R/W and /WE,
   between Vcc, CS2 and Vcc, and D0 through D7. If the address bus test
   fails, check A0 through A12.
Using 62256 chips instead of 6264
---------------------------------
You can freely substitute 62256 chips for the 6264. Only two pins differ:

             6264 RAM                          62256 RAM
                ___ ___                           ___ ___
   NC   1  =|   V   |=  28  Vcc     A14   1  =|   V   |=  28  Vcc
  A12   2  =|       |=  27  /WE     A12   2  =|       |=  27  /WE
   A7   3  =|       |=  26  CS2      A7   3  =|       |=  26  A13
   A6   4  =|       |=  25  A8       A6   4  =|       |=  25  A8
   A5   5  =|       |=  24  A9       A5   5  =|       |=  24  A9
   A4   6  =|       |=  23  A11      A4   6  =|       |=  23  A11
   A3   7  =|       |=  22  /OE      A3   7  =|       |=  22  /OE
   A2   8  =|       |=  21  A10      A2   8  =|       |=  21  A10
   A1   9  =|       |=  20  /CS1     A1   9  =|       |=  20  /CS1
   A0  10  =|       |=  19  D7       A0  10  =|       |=  19  D7
   D0  11  =|       |=  18  D6       D0  11  =|       |=  18  D6
```

```
   D1  12  =|       |=  17  D5       D1  12  =|       |=  17  D5
   D2  13  =|       |=  16  D4       D2  13  =|       |=  16  D4
  Vss  14  =|       |=  15  D3      Vss  14  =|       |=  15  D3
           `-------'                          `-------'
```

Pin 26 (A13) can be connected to Vcc just like on the 6264, but we need to wire pin 1 (A14) to either Vcc, Vss or another address bus pin. It's probably
easiest to wire it to pin 2 (A12). We won't be using the greater capacity of the 62256 using this method (doing so would require some kind of decoder logic) but sometimes 62256 chips are cheaper than 6264, or maybe you just have
some lying around.

Adding Block Select and Write Protect Switches

Adding block select switches allows you to chose which blocks should be populated on the fly. This allows you to chose between a basic expansion and a
"cartridge emulator" that allows you to load cartridge images into ram and run
them. I added one switch for each chip, giving the first chip the option between block 1 and 3, and the second between 2 and 5.

```
  block 1 -----o
               \
                o----- to /CS1 and /WE on chip 1

  block 3 -----o


  block 2 -----o
               \
                o----- to /CS1 and /WE on chip 2

  block 5 -----o
```

As a kind of copy protection, some cartridge images try to modify themselves to detect if they're running from RAM. If we add write protect switches we'll
be able to run those as well. Switch to write enabled while loading, then switch to write protected and reset.

```
      R/W -----o
               \
                o----- to /WE on chip 1

     +5 V -----o


      R/W -----o
```

```
                \
                 o----- to /WE on chip 2

     +5 V -----o

+5 V is the same as Vcc.
```

Quick Quadratic Splines

```
S. Judd <sjudd@ffd2.com>

Splines are neat.

Splines are a way of drawing curves -- specifically, drawing a curve through
any set of _points_ that you choose; for example, a curve that starts at
(0,0),
goes over to (100,50), then loops back to (50,0), and continues on through
any number of points can be drawn just by specifying the points.

This is a very useful thing!  One place splines are used is in drawing
fonts.
With a relatively small list of points, it becomes possible to draw letters
in arbitrary shapes (and scale those shapes easily).  Another application
is animation -- it is possible to specify a long motion path using just
a few points along the path.  Furthermore, an animated object might have
different parts that move differently -- arms, legs, head, etc.  By just
specifying a few frames of the animation, splines can be used to generate
the in-between frames.

Chances are good that by now you are thinking about some of your own
applications for splines, so let's see how they work.

Most articles/books/etc. that talk about splines talk about "cubic splines".
This article is going to talk about "quadratic splines", which never seem
to be mentioned.  Which is a pity, since quadratic splines are perfectly
adequate for many (if not most) spline applications and are far more
computationally efficient.

Graphically, a two-dimensional quadratic spline looks something like:

              * P1
             / \
            /   \
           /     \
          /  . .  \
         / .     . \
        /.         .\
       /.            .
      /.              * P2
```

```
     .
    .
P0 *
```

(Another brilliant display of ASCII art). The curve starts at P0, moves towards P1, and then turns around and heads towards P2, where it ends. If we draw lines from P0 to P1 and from P1 to P2, these lines are tangent to the curve at the endpoints P0 and P2; that is, these lines say what "direction" the curve is going at the endpoints. Using the above diagram, it should be very easy to visualize how changing P1 changes the shape of the curve.

So here, for your computing pleasure, is a quadratic spline:

```
P(t) = P0*(1-t)^2 + 2*P1*t*(1-t) + P2*t^2
```

where P0, P1, P2 are constant values, and t ranges from zero to one. If you don't like equations, then how about a computer program to compute the above:

```
10 P0=10:P1=100:P2=37
20 FOR T=0 TO 1 STEP 1/16
30 PRINT T,P0*(1-T)*(1-T) + 2*P1*T*(1-T) + P2*T*T
40 NEXT
```

This is actually very easy to understand. When t=0, P(t) = P0. As t starts to increase, (1-t) starts to decrease and P(t) starts moving towards P1. As t gets even larger P(t) starts moving towards P2, until finally, at t=1, P(t) = P2.

You may be wondering why there is that 2*P1 in the equation, instead of just P1, but that will become apparent shortly.

At this point there is an important thing to notice about the above equations: adding _any_ constant to P0 P1 and P2 is just like adding the constant to the entire spline P. Let P0=P0+C, P1=P1+C, P2=P2+C; the spline is then

```
P'(t) = (P0+C)*(1-t)^2 + 2*(P1+C)*t*(1-t) + (P2+C)*t^2
      = [ P0*(1-t)^2 + 2*P2*t*(1-t) + P2*t^2 ] + C*[(1-t)^2 + 2*t*(1-t) +
t^2]
      = P(t) + C
```

This is one reason that factor of 2 multiplying P1 is important -- it makes the expression multiplying C add up to 1. This property means that splines can be _translated_ easily. Note that one value you can translate everything
by is P0 -- subtract P0 from each point and the spline becomes

```
P(t) - P0 = 2*(P1-P0)*t*(1-t) + (P2-P1)*t^2
```

```
and hence

P(t) = 2*(P1-P0)*t*(1-t) + (P2-P1)*t^2 + P0

which gets rid of the P0*(1-t)^2 term.  Depending on how the spline
algorithm
works, this can save some computation time.

Now, the points P0, P1 etc. are called "control points".  If you're on the
ball
(or if you've tried the BASIC program above) you've noticed that while the
spline starts at P0 and ends at P2, it never actually reaches P1 -- it just
heads towards it but then heads away towards P2.  If a different P1 is
chosen, the spline still starts at P0 and ends at P2 but takes a different
path between the endpoints.  P1 controls the shape of the curve.
So now let's say we want a curve that passes through three points, s0, s1,
and s2, using a quadratic spline.  P0 and P2 are easy to choose:

    P0 = s0
    P2 = s2

P1 can actually be chosen at this point to be any number of points.  All we
do is choose a value for t -- call it t1 -- when the spline should hit s1:

P(t1) = s1 => P1 = (s1 - P0*(1-t1)^2 - P2*t1^2) / (2*(t1*(1-t1))

(just substitute t=t1 into the equation for P(t) and solve for P1).  So if
t1=1/2, say, then

P1 = (s1 - P0/4 - P2/4) / (2 * 1/4)
   = 2*s1 - (P0 + P2)/2

and the spline will hit s1 halfway through the iteration at t=1/2.

In a typical application, of course, the curve will need to go through tens,
hundreds, even thousands of points.  Naturally, the way to do this is by
joining a bunch of splines together.

If we just choose a spline for every three points, using the above equation
for P1, there will typically be sharp corners where the splines join
together.  Sometimes this is a good thing -- for example, when drawing
something like a valentine we might want sharp corners.  But usually
what is needed is a nice smooth curve through all the points.

With quadratic splines this is a piece of cake.  Each spline has three
control points: the two endpoints, and the middle control point P1.  The
endpoints are simply chosen to be the points we want to pass through,
and P1 can then be chosen to make all the connections smooth.

That is, given a list of points s0, s1, s2, ... to put a curve through,
choose s0 and s1 to be the endpoints of the first spline, s1 and s2 to
```

be the endpoints of the second spline, and so on, and choose the middle control points to make everything smooth.

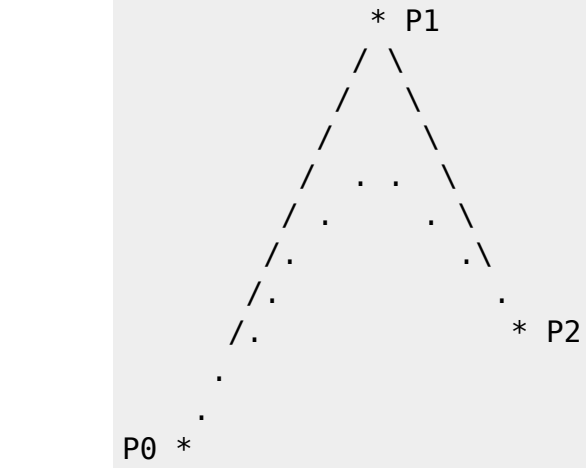
"Smooth" here means a continuous first derivative -- if you don't know what a derivative is then don't worry about it, it just means that the "slope" of each spline is the same where they join together. Given a spline

S1(t) = P0*(1-t)^2 + 2*P1*t*(1-t) + P2*t^2

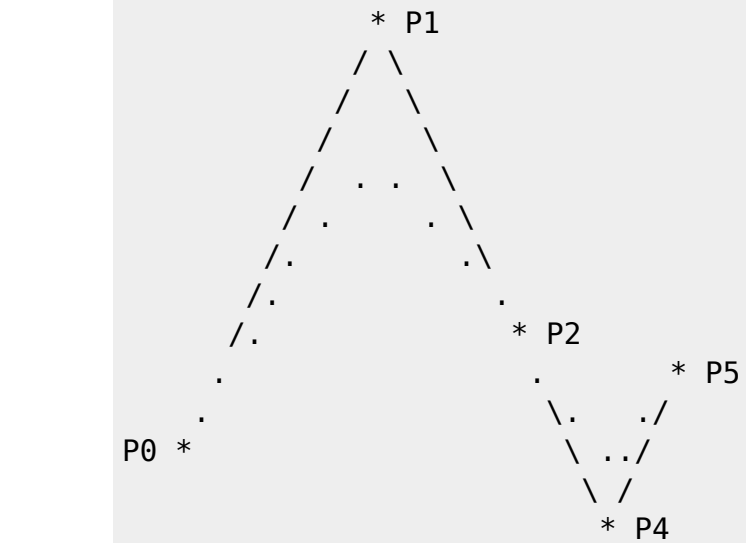
the derivative at t=1 is given by

S1' = 2*(P2 - P1)

P2-P1 is simply the line segment running from P1 to P2, which verifies what was said earlier: at P2 the curve is tangent to a line drawn from P1 to P2.



P2-P1 is the direction the curve is headed in. So to join another spline smoothly to the first one, simply extend this line segment and make sure the second middle control point lies somewhere on that line:



Since the first spline is tangent to P2-P1, and the second spline is tangent to P4-P2, the two splines have the same slope at P2, and join smoothly. Note that P4 can be _anywhere_ along the P2-P1 line, and choosing different

```
values for P4 will change the curve.

To make this more precise, let the two splines be given by

S1(t) = P0*(1-t)^2 + 2*P1*t*(1-t) + P2*t^2
S2(t) = P3*(1-t)^2 + 2*P4*t*(1-t) + P5*t^2

To join them smoothly, the slope of the first spline must be proportional
to the slope of the second spline at the joint:

P4-P3 = c*(P2-P1)

where c is some constant (the _direction_ of the two slopes need to be the
same, but not the magnitude).  So, to make the second spline fit smoothly
to the first we simply choose the middle control point P4 as

P4 = P3 + c*(P2-P1)

or, in plain English, starting from P3 (the joining point) move some
distance c in the direction P2-P1.  To see why the factor c is important
just scroll up a page or two to the diagram, and imagine how the curve
changes as P4 slides back and forth along the line.

We now have the tools to construct a smooth curve that passes through any
points s0, s1, s2, s3, ...  Let the first spline pass through s0, s1,
and s2, using the t=1/2 formula given earlier:

P0 = s0
P1 = 2*s1 - (s0 + s2)/2
P2 = s2

P1 could be chosen in other ways, of course, but the above usually works
pretty well.  Then choose each succeeding spline to match up smoothly with
the previous spline:

P3 = s2         ;start at spline 1 endpoint
P4 = P3 + c1*(P2-P1)    ;smooth joint
P5 = s3         ;next point

P6 = s3
P7 = P6 + c2*(P5-P4)
P8 = s4

and so on.  (When drawing smooth curves, it is of course really only
necessary to store the middle control points P4 P7 etc. for each spline).
The choice of the constants c1 c2 etc. really depends on how you want the
curve to look.

As you can see above, each extra point requires another spline.  One spline
per point may sound like a lot, but there are a lot of pixels in-between
each
```

```
pair of points, and more importantly these are quadratic splines and hence
can be made to go very, very fast.

Computing the spline
--------------------

Note that the control points are one-time calculations; those three constant
values completely determine the spline.  One way of drawing the spline is
to use some lookup tables for t^2, 2*t*(t-1), and (1-t)^2 and do a little
fast multiply magic.  This is fast and straightforward, so I won't talk
about it much except to mention that signed multiplies may be required,
and each curve is limited to 256 points (t can go from 0 to 255 in the
lookup tables).  Note also that it would be nice to have an "incremental"
plot routine, i.e. one that just updates pointers when necessary, instead
of recomputing all the bitmap pointers at every iteration.

Which of course brings us to a second way of drawing the spline: to ask
"how does the spline change when I increment t?"

Let's say that t is incremented by dt at each step (t -> t+dt).  At each
step, then, the spline changes by

S(t+dt) = P0*(1-t-dt)^2 + P1*2*(t+dt)*(1-t-dt) + P2*(t+dt)^2

    = S(t) + k1*dt^2 + k1*2*t*dt + k2

where k1 = P0 - 2*P1 + P2 and k2 = 2*dt*(P1 - P0).  That is, to advance one
step, we add

    k1*dt^2 + 2*k1*t*dt + k2

to the current value.  Take a look at how the above value changes as t
changes:

t   k1*dt^2 + 2*k1*t*dt + k2
--  ------------------------
0   k1*dt^2            + k2 =   k1*dt^2 + k2
dt  k1*dt^2 + 2*k1*dt^2 + k2 = 3*k1*dt^2 + k2
2*dt    k1*dt^2 + 4*k1*dt^2 + k2 = 5*k1*dt^2 + k2
3*dt    k1*dt^2 + 6*k1*dt^2 + k2 = 7*k1*dt^2 + k2

and so on.  This means that all we have to do to advance the spline is
something like

    c0 = k1*dt^2
    C = c0 + k2
    S = P0
:loop
    S = S + C
    plot S
    C = C + 2*c0
```

```
    loop

Which is really, really fast.  Instead of specifying the spline with the
three values P0, P1, and P2 we can specify it by the three values P0, k1,
and k2.

It is possible to understand this iterative method somewhat.  The update
constant k1 can be written as

    k1 = (P0-P1) + (P2-P1)

whereas k2 goes like

    k2 ~= (P1-P0).

P1-P0 is the "direction" line segment towards P1, so the curve starts moving
towards P1.  But k1 starts to balance that out with the P0-P1 term (the line
segment pointing in the opposite direction), while moving towards P2 with
the
P2-P1 term, which is the direction line segment from P1 to P2.  Now, isn't
that all nice and clear now?

The Program
-----------

At the end of this article I've included a sort of "spline laboratory" --
it's a little BASIC program that lets you experiment with different aspects
of splines and see what they look like.  It uses BLARG for the graphics, so
I've included that as well.

When you run it, it plots three points on the screen to draw a curve
through.
You can move these points around, as well as add new points, draw the
splines,
draw the control point "a-frames", and so on.  It has a ton of commands,
basically because I just added them as I got curious about different things:

+/-     Move to next/previous point
cursor keys Move current point
return      Draw spline
space       Clear screen
*       Toggle point display
< >       Decrease/increase accuracy (number of bits in iteration)
1 <-     Increase/decrease time step (number of points in spline)
=       List control points (numerical values)
.       Draw control point a-frame
S       Toggle smooth splines (see below)
A       Add more points
Q       Quit

Pressing "A" adds three points at a time, starting at the last point in
```

```
the curve.  The reason for adding three points, instead of just one, is
because of the "S"mooth spline feature.  When smooth splines are selected,
the program will draw a smooth curve through all the points.  When toggled
off, the program will draw a spline through every three points, using the
"t=1/2 method" to select the middle control point.

The program uses the fast iteration method described earlier, using fixed
precision arithmetic (since this is what an assembly program would do).
The < and > keys are used to change this precision, since certain splines
need more precision than others.

Finally, the value CC=0.4 is set in line 10.  This is the "smoothing"
constant, i.e. c1 in the equation for P4 below:

P3 = s2         ;start at spline 1 endpoint
P4 = P3 + c1*(P2-P1)    ;smooth joint
P5 = s3         ;next point

You might want to experiment with different values, to see what happens
(or else let each spline have their own value).

Anyways, the program is not meant to be terribly profound -- just a starting
point and something to play with to work out your own ideas on!

Cubic Splines
-------------

Just for completeness, here is a cubic spline:

P(t) = P0*(1-t)^3 + 3*P1*t*(1-t)^2 + 3*P2*(1-t)*t^2 + P3*t^3

As you can see, it has four terms instead of three, and is cubic in t.  But
the idea is the same -- at t=0 it starts at P0, as t increases moves
towards P1, then towards P2, and finally ends at P3.

You can also see that it is significantly more computationally involved than
the quadratic spline.  Moreover, computing the middle control points P1 and
P2
is also fairly involved -- much more involved than with the quadratic
spline.

So, why use a cubic spline?  Two reasons: first, you can specify the
derivative
at _both_ endpoints -- the direction line P1-P0 will be the slope at the
starting point, and the direction P3-P2 will be the slope at the end point.
This gives some flexibility needed for certain applications.

Second, you can alternatively match _second_ derivatives at the connection
points, which again can be useful for certain applications.

For more information about cubic splines, just check out any decent graphics
```

```
book.

The Program
-----------

begin 644 slab.zip
M4$L#! H    & $]LQQJSS3>D,0<  /0(   *    8FQA<F<N;6TN;P\  0(4
M!18%!@T(+0=]_OY^"1$3%A,&!??W]U<#(FR(L/E!SS)AVIRI4]2S4P,LW#]U
MPL+^4\_'F4[Y_:DAIF.>?VK@A0>"!DA88.S^D04BAY[?O'_S XWSAQZ?]#YH
M1@S\/_3[H@OV)CBP<T?@RB'>9PH0^-2G+$%F?5H5Y-6@3JF"# IR:M&A4HL@
MW[3HU*E!CQ;1OBC2GR"=!I7J-&E0%L> @\R_M*A+D$R?6BDV5:1)G2XQ?BK4
MHD&,+;0(E)DND?X(=D\Q%8U3B88I'W0J%6F4:%(KT:TSU:)2HJN^*%2D19U$
MM;W2JD2)6%]5*XCG_@<9\P5MO\QY(W=;!-Z)]W(W>L=;@SOF&ISIWN,%[P(_
M#[Q&<B,7A=+X5,$SF7+XH4FE#F5:5/Q1J9''#?_4J%'R39\2+5H>=M(CW)O#
MUF"?%UJENJ[13^D;:OJA3YG"5@^P:W#XP?T/_VL\O7&2Q\4?GVU\]G'IQR4>
M!\D1..B"A/^#_ P<]D*0PH%#IY^Y?\#4_\&:_M]Y?T#TN88S-OSSR"'2FW32
M_WN/! X?8W+PL"MFHQXZ0>3HH;M')RWH(>-3/CQ&,. C@S=\CLD?0X?@J_<U
MXPZ+[X=N'[[YY"#ABT\N$KYZ,.V>F7P][)<D'\)<D<G?51-. LG*X2]8#PFI
M9'G)AFD0I#2FH==/#AL^A#<?))TTY7^V'X$9AW _G?K)ME-+ <>I@YP+$(&]
M-439(@J'$\K=]>FDX!6J[BT6.-+>889GXCU*IN=/=;Q!KK]=FB+.=A?GPA1A
M%S1>$:9%G(K;A9PA/U0-'CEL!=DG!UJZ8OJ D ."59P2W\VO5*G2MGR-[;P!
M&NRO0A@M6[&L/^BZ/XB*/?$QENU7/F0!/Q^S#ZINE7$,LK2_KN8SK</'FYIF
MX0 0ZTW!!^\9/G@4C#6#Y1O/8P7['S!0D! 1$A08\/_?GQ\?'AP@.1LF '1A
M 7I5+(/L[^J9"Y86B+,\1=6E!Y:O"%MC:8*D"Q;:M'&3%DCHG >22K\BX8&0
MM1=&:?BD:&3?K5\0ML&D_],&%(VXL$G'Z(4[QCRL-+#').@5$1;^![W1_VJ$
MT<*C44(NF&SXAF!NI_%(=<72HTL7Q*%PA;9J!+>Y<"GO5*8S0J:8$)!RKCDA
M<-4P$28$MXRK>K^1:H,T'-Q=\;W3Q3] \E;#39LD?Q)R'/RAMX'R@,"LM*@
M[15#^]DK\?.HR8:YP.'B)04T7$8.'"$DU^Z%^O]ZSWZ*RJ%OXP92N6A9DT8%
M0C=OANA]_<-TF/PD;/2E^Z=%$'-0<Z^_;_[3\)4#-'U"[L8PAXU=0.V!SI +
M;8'P+Y8W"?F&]_6<#1'>\R!/#XM"#D^9_U7>YY)"STD!_)^IT>'&"T(L7O9$
M[$:9!OT"[R[5SO]>YU;_BIV9*AWV&/Y =KR@P)(#B]B.@]'IAI_A8+E4=B[K
M&#:6M Q$JXC=I<GLS9<&"_H ]O UPJR8_ _-?&H,"B?O<):^\73?-O8]6K1,
MKP[KA4OQ"A_E*YP4(%8"MQI+1PG;H:I9H8H@]>G"IMC$Z@RL.2$.;#%=H>#9
MT]%N#;^1N!<T79J)JJ6LE_*9^X1.]F"*<=7)D4-&B!%5)CBF+!=&50D8RD<)
MIRU$7<1?>&UA-J;W<Z)YV2SS,[@N-83(NEL,,C?B$:I:V9S*\C8VI]K8!,0%
MY$T!WA4L-E[:K1*:?L-MS;<UFN;302(O;G"-L>$7$[HM[/5EW<"OT3C();WU
MYE)L5FS<K7&"$+YO6VQF&1E2(QOOM'Q;XFXA^S:JU@#COZE<\W!%Z7#S!4FA
M@YMV"S^$:@G5_K:WR\4Q ?:9%\OWFF2FQS=;V,$4-F9R:\U@:$OA=#&-&K[(
MY!E.)@MXE#PF/:K4WL=,.#\_9O/';!Q5IRRV&,F5Q+YT:I@>/!F^NY@SAD\D
MJC[1SWQ_#*Q ,9M/#89T2GE=QJ=,NIS&8/+>1_KF^6.(>:AB\Z?.O>2<4 Z;
M.B2;OL+)>X&P(I('X@A+62CM E#:5ML4R_BI;ZM(N1=5P[%J\!O45LR,_M-Y
M(BS!)#W!V/NDT?V1]X.!PN!?6Y0-/"B%;I1B9 KPH *D40$Y\6#! _P/L'5$
MZ"E!0F_96F#WA ,;$=\:8'>D*9#W:QQM2OX-^QIOY'QKC- ;]>2GQMGHN'4G
M!-F(>)!Y <*'$?-B";E53Q-@4QBI=3 OFH#)8.C[3BMAW'XKA(I%@\04#063
MAKU:&G!*O@W]B&%?=:$IOW493)8R'[Y!<"S9!47_=]3Z8"-$LT#(ZL%#A#[3
M44%R:M"-NH)$GMO/PJ%/GIGZ<*&%>9JI2:1ND/&F1]P8L8#M;I^.[GH:O 2,
M-'N#G'ELFN],(V?74&)P2@Q R:D',$$_,>$4F. $F&#XR./DN%OMG\QJBJ!!
MQ4P?9"8T= &/KT2408-GM#R8;10!:$X:H#.Y4T8@DA'Y" P&4$L#! H    &
M %1LQQJZ:;<U#@8  %P)   )    <W!L:6YE;&%B%!$2! ($!08%#@8.!PD(
M/@<>!W[__PT1$P<&! ,&!006]_?W)P,B3(BP(,#R,6T"#(E0(4"B5&7'U#DU
MJ=8BQK>I<^A0V3!=T@0X%.%#@(9!8AV*TB;-E"RS&.-0L1CC68P=H%:$% $^
```

```
M!NFT9TR9-W4>?3JUJE"0,F'"A D0,$(>_'LJ%*/K\"_,%,=XC./1#]EE(!]'
M^2\C# K0* SS!'UJTZ=/J2(%:91IT*, >R,T"M!)_F7JS)*O,9Z6[#'D#=GV
M$_KXGQSA4H#C8]Q=I_JA ,\CA HP<)*QD[*_3G&O>Q+7I3C>%PSP/T*R -6#
MA)&-0\F6XLH$J#&A6X"A@Y+4N1@$\64? _O,E"!/@VP< [ "O)D0+P!K%=E9
M!'7N55:QKHX)D&I"0=S<@1NX*N['3<B(FR-QVYK*SI+/CA-:XN8(W,#R^SGA
M)NYM.XAMOUD3(.^$G=I9I<A3#?>K45"R*5RKSD0[<ZDG!$59AU &\/N$H^C9
MAY%!*L3.2OL*%9;29A[:H*VT'U+A*6V.H6T+)\*3O#@5IK+F+]8PPWI5H2IK
M7F%MJCR4]17<0XW5<W\*U@*TJY"6-[?PYE7>/AM&=\0/,+'"6][<P]MGN-QR
M5IA+FJ-(>\Z<. %N5[A+FEV*L*08'!N5G!H>Q"&:Q!":9!*:1/ U8\>9M?(5
M]O+G*?Y6\KZ*^  ++O1EK;-CBPQR;0BW2Q<B:WD3ONH97]95&09O!"^5&G0P
MSE7C9.7\5>/A^>O >*FE9<O:'+E5ENI"D0$3T=+%+F09D+S.S#I@EX/UA38#
M8 +-##(I"TU 2*_GA3NZ>4JW2 ;I;NR%0 -X0@HS@/,9EUC&^D*E <>]'<16
M 19?:'X V@$#E+[P_("?=7".'FKXJ0"[+T0_L#$(/4V_4/Y )3932,Y@^?L+
M^0\,#"*M($FMC >HDJ%!@EAEHZ0:XZRLYJK$LU.5L8SXS&G^C&;8:T.^DX_S
MK@PA$LPB#\\B#\TB#YI%'CPM0XT$*C\6UJ98 S,D26#Q*PFYX;E7.FMA7LZ-
MZ'NLAQFZ).@8(&F&2PD$]J%/FT*M2K7(<9U2E?J4R:=_F@1YD9TU0R;Z4;]S
MAE8)K"Z3M.+K(%P*:<O1,M*NL2&8<);):EQ7"32O8<LI^ZK0<VRI)Q\9?Y9Q
M\"SCG&6< ?)LJ 9V"6@R!T3FO.%;@HLNWF$PYC%KY'JS#>G,$VE@J'2Y#?T2
M '487QVJ'6_#OP1U K3<$# !F0;VFJ7VJD^K3E%^5:E6BS"L-PPTWMH-!Q.
MY94L6=1CRW)B&DZ_WA"1Y@!.HCED^0 [.FQ,0%&6Y2GF4&RT)>NJ8E\MHV F
M<,T"#L\"KEG <8!%'4H"/ Y3/*[*: -\U>$EA+E)8$,RRHIUR DV5_):T3I\
MM5@#.X3U6HD694HU:,NF16;[4R*ML%GLUSN,Y5W)#FL3H 6W=JB; "U89],@
MI+B-B_N9-EN1M+92W"&QR@X#,@SPQY05&BE?IJQ4T,PTALY0LG"?B/^Q7)](
M(9[QH;-ZS5*/9JD'SU(/FJ5>L]3KTH?80E;4\"S\3 5Z4BN>\\2OD\CR.F\K
MY&9!Q@&"?2B>0,B$RJ#](%.QU 6H]B$[@84#O',0O0!]A(>RCL5F^,."!6+F
M+3_6OEE4ANVKEJM#+"\W'T-EXGCX"W^HP%QGPQA'58WSJG$\Q\O\YP\I%A@\
MPKL:V>T_] #O6:Y@V'8EM.<*G_&(:H2(PERN,'X9IRL,I7.8$%L6@"W"LUF/
M,9L);V,M9#I/B#(+N#2+#*<*D6<!B0$ZA5>7D5^M$(T6N/5K(3X_$.LL]I-H
M[ =ETDMG#!'Z 8I^!.5;U=3CZ)\"H:"L??R&1WRGJY/4[$/4?L"CD[53926\
MA'.(Y ^(=)(R!EIW0W1_H*136<X%.!PB_H."0+[;(>)>L-.#3AGYV?2^'Z+P
MA41>XBGR-E<A\CJ#;)KZ(9NG4=D3/D"*$9E]KWK8NXB7LPLC9F^G?4Q5%&<.
M^S,C6L->>5*_3)"J",VW[.03'T4SI@*<&A'<O==& 1Y+F&)C3W-E-., LT94
MO]#1AA502P$""@,*    !@!/;,<:L\TWI#$'  #T"   "@
MI($     8FQA<F<N;6TN;U!+ 0(* PH    & %1LQQJZ:;<U#@8  %P)   )
M             "D@5D'  !S<&QI;F5L86)02P4&      (  @!O    C@T
#

end


.......
....
..
.                                    C=H #20
```

Main Articles

VIC KERNAL Disassembly Project - Part IV

Richard Cini
February, 2001

Introduction
============

In the last installment of this series, we examined the six remaining routines that are called from the IRQ and NMI vectors. The routines examined are responsible for updating the jiffy clock, determining the location of color RAM, scanning the keyboard, and reading and setting the cursor position.
Having fully completed the main processor vectors, we'll continue this series by examining other Kernal routines.

More Subroutines
================
Much of the last three articles dealt with routines that were used in the VIC's startup process, configuring the screen and I/O devices, and polling
the keyboard for user input.

Although the entire Kernal ROM consists of probably over 100 individual subroutines -- all indirectly callable in some way - the VIC's developers really intended the user to only call the 39 routines accessible through the jump table at the end of the ROM. When looking at this table, we've so far only discussed only 10 of the routines contained therein.

It was designed in this fashion for multiple reasons. The functions selected for the jump table were clearly the ones Commodore found to be most useful for software programmers. The other routines were "internal" routines used by the "public" routines and the BASIC ROM.

The jump table also hid code changes from the software programmer, although I
don't believe that there were ever any significant updates to the Kernel after
its initial release. The calling routine and parameters remained consistent even if the underlying code changed. It also facilitated sharing programs between machines with common lineage. For example, certain programs could be shared between the PET, VIC and the C64. Of course, minor modifications of these programs were required to account for different I/O locations.

This structure is no different than a modern operating system such as Windows,
or even the Macintosh, which had the concept of a ROM "toolbox."

Anyway, I digress...

```
When examining the functions available in the jump table, the following
distribution becomes apparent:

User I/O (screen/keyboard):  3 functions
Timekeeping:             3 functions
Memory management:       3 functions
Processor control:       4 functions
Device I/O:         26 functions

Commodore appeared to be giving greatest support for interfacing the VIC to
various devices. Many of these routines were used by the built-in BASIC
interpreter to support file I/O through the tape deck and the serial IEEE
port, as well as printer and RS232 support. Unfortunately, the only devices
that made it onto the IEEE bus were floppy drive and printer devices. If
there
are others, I appreciate knowing about them.

    Anyway, this provides a nice segue into looking at the device I/O
routines in some detail.

    The first step is opening a device. All devices are eligible to be
opened in this manner. The OPEN function requires that the program call two
preparatory routines first. The first one, SETNAM sets the filename
parameter
required by OPEN. Clearly only certain devices make use of a filename (such
as
the cassette or disk drive devices). For devices not requiring a filename
pointer, just set the pointer to "null".


FE49  ;=================================================
FE49  ; ISETNM - Set filename (internal)
FE49  ; Call with .A = filename length, .X = LSB of filename
FE49  ; string, .Y = MSB of filename string
FE49
FE49  ISETNM
FE49 85 B7  STA FNMLEN        ; set length
FE4B 86 BB      STX FNPTR        ; ptr L
FE4D 84 BC      STY FNPTR+1        ; ptr H
FE4F 60     RTS

    Calling SETNAM sets three Zpage variables that tell the OPEN routine
where in memory to find a string containing the filename. Setting the length
parameter to zero yields a null string.

    When opening an RS232 channel, the filename string would contain two
ASCII characters representing the configuration values for the command
register and the control register.

    The next call that is required is to set the underlying device
parameters for the OPEN call.
```

```
FE50    ;=================================================
FE50    ; ISETLF - Set logical file parameters (internal)
FE50    ; Call with .A = file number, .X = device # (0-30),
FE50    ; .Y = command
FE50    ISETLF
FE50 85 B8      STA LOGFIL        ; file #
FE52 86 BA      STX CHANNL        ; device
FE54 84 B9      STY SECADR        ; secondary address
FE56 60         RTS
```

The SETLFS parameters are fairly common: the file handle number, the device to access, and any secondary address to send to the device. The combination of SETNAM and SETLFS is analogous to this BASIC statement:

```
OPEN 2,8,1, "mydat"
```

where "2" corresponds to the file handle number, "8" is the device, and "1" is the secondary address. These numbers are ultimately stored in their respective data tables at location IOPEN_S3 in the OPEN routine.

Opening the RS232 channel does not require the use of a secondary address.

During file I/O, it's helpful to check the status of the current device by using the READST function. READST retrieves the value of the CSTAT variable. CSTAT is set through a call to ISETMS1 (at $FE6A) by various I/O routines based on device response.

```
FE57    ;=================================================
FE57    ; IRDST - Get I/O status word (internal)
FE57    ;
FE57    IRDST
FE57 A5 BA      LDA CHANNL        ; get current device
FE59 C9 02      CMP #$02        ; RS232?
FE5B D0 0B      BNE ISETMS+2        ; no...branch to OS messages
FE5D AD 97 02   LDA RSSTAT      ; get RS232 status
FE60 A9 00      LDA #$00
FE62 8D 97 02   STA RSSTAT
FE65 60         RTS
FE66
FE66
FE66    ;=================================================
FE66    ; ISETMS - Control OS messages (internal)
FE66    ; On entry, .A is message number. Bit7=1 for KERNEL messages, and
FE66    ; Bit6=1 for control messages. Bit0-Bit5 is message number.
FE66    ;
FE66    ; set flag for OS messages
FE66    ;
FE66    ISETMS
FE66 85 9D          STA CMDMOD        ;save message #
FE68 A5 90          LDA CSTAT        ;get status
```

```
FE6A
FE6A    ; set ST bits
FE6A    ISETMS1
FE6A 05 90          ORA CSTAT        ;twiddle bits based on .A
FE6C 85 90          STA CSTAT        ;save status
FE6E 60             RTS
```

The return values of the READST call have different meanings depending on what device is being accessed:

BIT	Cassette	Serial R/W	Tape L/V
0	Write timeout		
1	Read timeout		
2	Short block	Short block	
3	Long block	Long block	
4	Unrecoverable read error	Any mismatch	
5	Checksum error	Checksum error	
6	End of file	EOI line	
7	End of tape	Device not present	End of tape

So, after having set the file name information and setting the device access parameters, we're ready to actually OPEN the file. The following code is heavily commented, so I'll only expand where necessary.

The Kernal keeps track of the open files using a series of file handle tables. A maximum of 10 logical files can be open at one time. FILTBL keeps track of the logical file handle established in the call to SETLFS. SECATB keeps track of the secondary address associated with a given logical file number. Finally, the DEVTBL variable tracks which device number relates to the
file handle. These three tables mirror the variables in the SETLFS call.

```
F40A    ;=======================================================
F40A    ; IOPEN - Open file (internal)
F40A    ; Required prior calls:  FFBA/SETLFS and FFBD/SETNAM
F40A    ; No arguments
F40A    IOPEN
F40A A6 B8          LDX LOGFIL    ;get file number
F40C D0 03          BNE IOPEN_S1    ;F411 read from output file?
F40E 4C 8D F7       JMP IOERMS6    ;Yes, emit "NOT INPUT FILE" error
F411
F411        IOPEN_S1
F411 20 CF F3       JSR FIND    ;locate file # in table
F414 D0 03          BNE IOPEN_S2    ;file number not found, so move on
F416 4C 81 F7       JMP IOERMS2    ;found, so emit "FILE OPEN" error
F419
F419        IOPEN_S2
F419 A6 98          LDX COPNFL    ;get # of open files
```

```
F41B E0 0A          CPX #$0A    ;are there 10 files open already?
F41D 90 03          BCC IOPEN_S3    ;no, OK to open new file
F41F 4C 7E F7       JMP IOERMS1    ;more than 10, "TOO MANY FILES" error
F422
F422        IOPEN_S3            ; allocate file slot in table
F422 E6 98          INC COPNFL    ;add 1 to count of open files
F424 A5 B8          LDA LOGFIL    ;get file number from call
F426 9D 59 02       STA FILTBL,X    ;save file # in table
F429 A5 B9          LDA SECADR    ; get secondary address
F42B 09 60          ORA #%01100000    ;$60 make it a device command
F42D 85 B9          STA SECADR    ;save it as secondary and add it to the
F42F 9D 6D 02       STA SECATB,X    ; SA table for that open file
F432 A5 BA          LDA CHANNL    ;save device number to the
F434 9D 63 02       STA DEVTBL,X    ;  device table
F437
F437                    ;Special handling for certain devices
F437 F0 5A          BEQ IOPENRC    ; keyboard device? Yes, return
F439
F439 C9 03          CMP #$03    ;Is the device the screen?
F43B F0 56          BEQ IOPENRC    ; yes, then return clear
F43D 90 05          BCC IOPEN_S4    ;must then be the tape or RS232-branch
F43F
F43F                    ; Start IEEE stuff
F43F 20 95 F4       JSR SENDSA    ;send secondary to IEEE bus and
F442 90 4F          BCC IOPENRC    ; return clear
F444
F444         IOPEN_S4
F444 C9 02          CMP #$02    ;RS232 (2)?
F446 D0 03          BNE IOPEN_S5    ;not RS232, so process tape device
F448 4C C7 F4       JMP SEROPN    ;open RS232 device
F44B
F44B         IOPEN_S5           ;Tape device (1)
F44B 20 4D F8       JSR GETBFA    ;Get tape buffer address
F44E B0 03          BCS IOPEN_S6    ;MSB>2? No, continue with open
F450 4C 96 F7       JMP IOERMS9    ;Bad tape buffer, emit "ILLEGAL DEVICE
F453                    ; NUMBER" error
F453
F453         IOPEN_S6           ;Continue with tape processing
F453 A5 B9          LDA SECADR    ;get SA
F455 29 0F          AND #%00001111    ;Are we in write/save mode?
F457 D0 1F          BNE IOPEN2    ;yes, prompt for Play + Record
F459
F459 20 94 F8       JSR PLAYMS    ;wait for "PLAY" key
F45C B0 36          BCS IOPENRC+1    ;$F494 return CY=1
F45E
F45E 20 47 F6       JSR SRCHMS    ;print "Searching for [name]" message
F461 A5 B7          LDA FNMLEN    ;get length of filename
F463 F0 0A          BEQ IOPEN1    ;no name specified, so search for
F465                    ;  next header
F465
F465 20 67 F8       JSR LOCSPH    ;search for specific tape header
```

```
F468 90 18          BCC IOPEN3    ;$F482 go get header
F46A F0 28          BEQ IOPENRC+1    ;$F494 return found CY=1
F46C
F46C         IOPENA         ; done searching and file was not found
F46C 4C 87 F7       JMP IOERMS4    ;F787 "FILE NOT FOUND" error
F46F
F46F         IOPEN1         ; Get next tape header
F46F 20 AF F7       JSR LOCTPH    ;search for next header
F472 F0 20          BEQ IOPENRC+1    ;$F494 return found CY=1
F474 90 0C          BCC IOPEN3    ;$F482 go get header
F476 B0 F4          BCS IOPENA    ;$F46C not found
F478
F478         IOPEN2         ; write tape header
F478 20 B7 F8       JSR RECDMS    ;wait for REC & PLAY keys
F47B B0 17          BCS IOPENRC+1    ;$F494 return CY=1
F47D
F47D A9 04          LDA #$04    ;control byte ID for data header
F47F 20 E7 F7       JSR WRTPHD    ;write tape header
F482
F482         IOPEN3
F482 A9 BF          LDA #$BF    ;pointer to tape buffer head
F484 A4 B9          LDY SECADR    ; get secondary address
F486 C0 60          CPY #$60    ;SA=0 (read) mode?
F488 F0 07          BEQ IOPENRC-2    ;$F491 Yes, skip write
F48A
F48A A0 00          LDY #$00    ; write mode
F48C A9 02          LDA #$02    ;control byte ID for data block
F48E 91 B2          STA (TAPE1),Y    ;write it to buffer
F490 98             TYA
F491 85 A6          STA BUFPNT    ;save pointer
F493
F493         IOPENRC
F493 18             CLC
F494 60             RTS
```

The OPEN function spends most of its time saving parameter information into variable tables and figuring out which device to open. If the OPEN call is for the tape device, the code either reads or writes the respective tape header.

OPEN calls nine other routines, six of which deal with tape input and output. The three non-tape routines are FIND (find file number in table), SENDSA (send secondary address), and SEROPN (open RS232 device). Since the OPEN code spiders out so, I'm going to leave the tape-related routines to another article.

FIND is a very easy routine. It searches through the file handle table to see if the handle of the file we're trying to open is already being used as
a result of a previous call to OPEN. Clearly each file handle number has to be

```
unique to prevent confusion at the system level. If a file is already in use
with the same handle number, .X returns non-zero, enabling the calling
routine
to drop into an error handler ("FILE OPEN" error).

F3CF    ;=======================================================
F3CF    ; FIND - Look for logical file number in open-file table
F3CF    ; On entry to FIND1, .A=file#. On exit, .X=offset in file table
F3FC    ; to matching file number.
F3CF
F3CF             FIND
F3CF A9 00          LDA #$00
F3D1 85 90          STA CSTAT    ;clear Status variable
F3D3 8A             TXA        ; .X is logical file number copied to
F3D4                    ; .A for use in FIND1
F3D4             FIND1
F3D4 A6 98          LDX COPNFL    ;get #of open files to count through
F3D6
F3D6             FINDLOOP       ; loop
F3D6 CA             DEX
F3D7 30 15          BMI FLATRBX    ;$F3EE reached 0, then exit
F3D9 DD 59 02       CMP FILTBL,X    ;is this the one?
F3DC D0 F8          BNE FINDLOOP    ;$F3D6 no, try again
F3DE 60             RTS        ;return with .X= offset into table

    The FLATRB routine is not used in OPEN, but the FIND subroutine exits
through it. I don't know why this is since the FIND routine has a return
path
of its own. Anyway, FLATRB takes an index number into the open file table
and
sets the Zpage file variables according to the values pointed to by the
index. For example, if I call FLATRB with .X=0, the file variables will be
set
with the information stored in location [0] of each of the file handle
table,
the device table, and the secondary address table. So, one could select an
open file handle, use FIND to get the index number, and then use FLATRB to
set
the Zpage variables to the right values for subsequent use.

F3DF    ;=======================================================
F3DF    ; FLATRB - Set file values
F3DF    ; On entry, .X = offset in the file tables. Returns with
F3DF    ; Zpage file variables set.
F3DF
F3DF         FLATRB
F3DF BD 59 02       LDA FILTBL,X
F3E2 85 B8          STA LOGFIL        ;get file handle number
F3E4 BD 63 02       LDA DEVTBL,X
F3E7 85 BA          STA CHANNL        ;get device
F3E9 BD 6D 02       LDA SECATB,X
```

```
F3EC 85 B9          STA SECADR        ;get SA
F3EE
F3EE             FLATRBX
F3EE 60             RTS
```

After the OPEN routine determines that the file handle to be opened is unique, assigns the file variables to the various data tables, and determines
which device is being opened, the code forks. If the device is the keyboard (device 0) or screen (device 3), OPEN returns. If it's the tape deck (device 1), the code continues by searching for or writing a tape header using the information supplied in the OPEN call. If it's for the RS232 adapter (device 2), execution jumps to SEROPN to continue the opening process. Finally, if it's an IEEE device (device >=4), then OPEN calls SENDSA to shift out the secondary address. When SENDSA returns, OPEN returns to the original caller. If SENDSA returns with the carry flag set (indicating an error condition) and then returns, execution will ultimately fall through to an "ILLEGAL DEVICE ERROR" error.

```
F495    ;=======================================================
F495    ; SENDSA - Send secondary address
F495
F495             SENDSA
F495 A5 B9          LDA SECADR    ;get SA
F497 30 2C          BMI SNDSARC    ;$F4C5 less than 0, exit
F499
F499 A4 B7          LDY FNMLEN    ;get filename length
F49B F0 28          BEQ SNDSARC    ;$F4C5 no filename, just exit
F49D
F49D                              ; prepare to send filename string
F49D A5 BA          LDA CHANNL    ;get device    number and...
F49F 20 18 EE       JSR ILISTN+1    ;  command it to listen
F4A2 A5 B9          LDA SECADR    ;get SA
F4A4 09 F0          ORA #%11110000    ;$F0 make it into listen command
F4A6 20 C0 EE       JSR ISECND    ;sent it to IEEE bus
F4A9 A5 90          LDA CSTAT    ;check status variable
F4AB 10 05          BPL SENDSA1    ;$F4B2 OK, then continue
F4AD
F4AD 68             PLA        ;error, set stack for caller's caller
F4AE 68             PLA
F4AF 4C 8A F7       JMP IOERMS5    ;emit "DEVICE NOT PRESENT" error
F4B2
F4B2             SENDSA1        ; continue processing-send filename
F4B2 A5 B7          LDA FNMLEN    ;get filename length
F4B4 F0 0C          BEQ SNDSARU    ;length is 0, so send unlisten command
F4B6                    ; There is a filename, so send it to IEEE
F4B6 A0 00          LDY #$00    ; set loop counter
F4B8
F4B8             SENDSALP       ; output filename to IEEE bus
F4B8 B1 BB          LDA (FNPTR),Y    ;get character
F4BA 20 E4 EE       JSR ICIOUT    ;send it
```

```
F4BD C8             INY        ;get next one
F4BE C4 B7          CPY FNMLEN    ; at the end?
F4C0 D0 F6          BNE SENDSALP    ;no, then loop
F4C2
F4C2             SNDSARU        ;done sending filename, so send
F4C2 20 04 EF       JSR IUNLSN    ; unlisten command to IEEE bus
F4C5
F4C5             SNDSARC
F4C5 18             CLC
F4C6 60             RTS
```

SENDSA is a simple routine that checks for a valid secondary address and whether the OPEN command involves a filename. Then, the routine commands the specified device to "listen" and sends the secondary address and the filename to it. When completed, the code sends the "unlisten" command to the device and returns success. If there is a problem sending data to the device,
a "device not present" error is generated.

The next routine is SEROPN. SEROPN is called by OPEN when it determines that the device that's being opened is the RS232 port. Here's a quick story about RS232 support in the VIC.

Based on some information that I've seen on the Web and what's available in "Inside VIC" and the "VIC20 Programmer's Reference Guide", I've concluded that
the VIC hardware was originally designed to include a MOS 6551 ACIA communications chip. The 6551, with appropriate level shifters, would have provided true hardware support for RS232. But for cost reasons, board space reasons, or both, the 6551 was dropped and emulated in software and in hardware by using a spare port on one of the existing 6522 VIAs.

The Kernal includes a lot of code to manage RS232 FIFO buffers and to perform
the bit shifting through the User Port (port B of VIA 1). The User Port only provides TTL-level RS232; Commodore and others sold an adapter that provided the level-shifting hardware (typically TI 1488/1489 chips) and a DB25F connector. The circuit is very simple and could have been built by any resourceful hobbyist.

Without further delay, here's the code that completes the opening process for
the RS232 serial device:

```
F4C7    ;=============================================================
F4C7    ; SEROPN - Open RS-232
F4C7    ;  "filename" contains the initialization data for the
F4C7    ;  command and control registers
F4C7
F4C7             SEROPN
F4C7 A9 06          LDA #%00000110    ;set VIA DDR. PB[2:1] are DTR and RTS
```

```
F4C9 8D 12 91       STA D1DDRB    ; signals, respectively
F4CC 8D 10 91       STA D1ORB
F4CF A9 EE          LDA #%11101110    ;$EE. Set PCR for CB2/CA2 manual high
F4D1 8D 1C 91       STA D1PCR    ; and CB1/CA1 for H->L IRQ trigger
F4D4                    ; CB2 is RS232-TX and CB1 is RS232-RX
F4D4 A0 00          LDY #$00
F4D6 8C 97 02       STY RSSTAT    ; clear status byte
F4D9
F4D9             SEROPLP
F4D9 C4 B7          CPY FNMLEN    ;is the filename length = 0?
F4DB F0 0A          BEQ SEROPN1    ;$F4E7 yes, go straight to open
F4DD
F4DD B1 BB          LDA (FNPTR),Y    ;copy first 4 chars of filename...
F4DF 99 93 02       STA M51CTR,Y    ; to buffer and loop
F4E2 C8             INY
F4E3 C0 04          CPY #$04
F4E5 D0 F2          BNE SEROPLP    ;$F4D9 loop
F4E7
F4E7             SEROPN1        ;done copying init_data
F4E7 20 27 F0       JSR BITCNT    ;get number of data bits
F4EA 8E 98 02       STX BITNUM    ;save data bits count
F4ED AD 93 02       LDA M51CTR    ;get control register
F4F0 29 0F          AND #%00001111    ;$0F isolate baud rate bits
F4F2 D0 00          BNE $+2        ;F4F4
F4F4
F4F4                    ;convert baud rate bitmap to clock
F4F4                    ; divisor
F4F4 0A             ASL A        ;*2
F4F5 AA             TAX
F4F6 BD 5A FF       LDA R232TB-2,X    ;$FF5A,X baud rate H
F4F9 0A             ASL A
F4FA A8             TAY
F4FB BD 5B FF       LDA R232TB-1,X    ;$FF5B,X baud rate L
F4FE 2A             ROL A
F4FF 48             PHA
F500 98             TYA
F501 69 C8          ADC #$C8
F503 8D 99 02       STA BAUDOF    ;save baud rate divisor
F506 68             PLA
F507 69 00          ADC #$00
F509 8D 9A 02       STA BAUDOF+1
F50C
F50C AD 94 02       LDA M51CDR    ; command register
F50F 4A             LSR A
F510 90 09          BCC SEROPN2    ;$F51B
F512
F512 AD 20 91       LDA D2ORB    ;check DSR (data set ready)
F515 0A             ASL A
F516 B0 03          BCS SEROPN2    ;$F51B ready, continue
F518 4C 16 F0       JMP DSRERR    ;not ready, DSR error
F51B
```

```
F51B              SEROPN2        ; device ready
F51B AD 9B 02        LDA RIDBE    ;set RS232 buffer pointer
F51E 8D 9C 02        STA RIDBS    ; -- RX --
F521 AD 9E 02        LDA RODBE
F524 8D 9D 02        STA RODBS    ; -- TX --
F527
F527                     ; prepare to "hide" buffer in memory
F527 20 75 FE        JSR IMEMTP+2    ; get MEMTOP
F52A A5 F8           LDA RIBUF+1    ;has RX buffer been created?
F52C D0 05           BNE SEROPN3    ;$F533 yes, check on TX buffer
F52E 88              DEY        ; else create input buffer
F52F 84 F8           STY RIBUF+1
F531 86 F7           STX RIBUF
F533
F533              SEROPN3
F533 A5 FA           LDA ROBUF+1    ;TX buffer created?
F535 D0 05           BNE SEROPN4    ;$F53C yes, skip create
F537 88              DEY
F538 84 FA           STY ROBUF+1    ;else create TX buffer
F53A 86 F9           STX ROBUF
F53C
F53C              SEROPN4        ; set MEMTOP to hide buffers from BASIC
F53C 38              SEC
F53D A9 F0           LDA #$F0
F53F 4C 7B FE        JMP STOTOP    ;$FE7B set new MEMTOP
```

SEROPN begins by extracting the values of the command and control registers from the "filename" string passed to it from the OPEN routine. SEROPN then saves the number of data bits contained in a serial data frame and then calculates the proper clock divisors for the selected baud rates. The 6551 transmit and receive clocks are emulated by using one of the timers in VIA1 to
shift out the data at the specific bit rate. Receiving bits is accomplished through the NMI routine discussed in Part 2 of this series.

In a nifty trick, SEROPN allocates transmit and receive buffer memory at the top of BASIC RAM and lowers the top of RAM variable to fake BASIC into thinking that there's 512 bytes less of memory.

The SEROPN routine finally exits and returns to OPEN by a JMP to STOTOP, the routine used to set the top of system memory.

Well, that's all that we have time for. Next time, we'll tackle some of the tape routines related to the OPEN command.

....
....
..
.

C=H #20

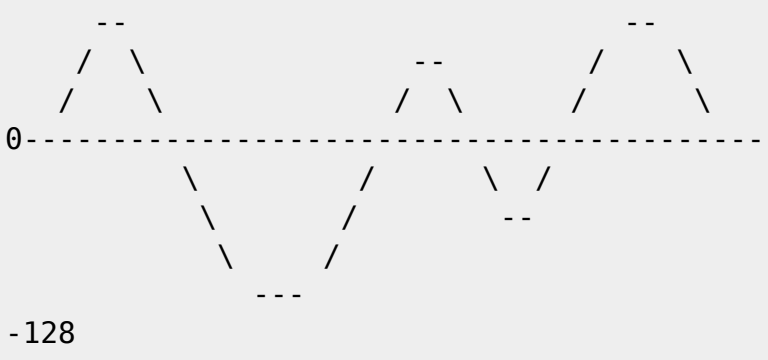
Mods and digital mixing

```
by Jolse Maginnis
jmaginni@postoffice.utas.edu.au

As most of you know, MOD playing on the C64 is nothing new, Nate/DAC wrote
modplay64 quite a while ago now, but there has never been any in depth look
at
how a mod is played. So in this article I will discuss everything you need
to
know about MODs and digital sound mixing (well the main bits anyway!)

Very recently I got sick of working on OS stuff and started working on my
own
modplayer for JOS, called, wait for it... Josmod! I was really quite
surprised
at the quality of sound it produced, particularly as it was only playing
with
4 Bit Mono SID digis.

At that stage Josmod only played 4-channel Amiga format MOD's which have
been
around for a long time but are now being outdated by newer formats such as
S3M, XM and IT. These new formats are in the same style as the old Amiga MOD
formats, except they allow for more channels, more instruments/samples and
different effects. Since the SCPU is a fast enough machine to mix and play
more than 4 channels at once, and I was sick of seeing so many MOD's that it
couldn't play because they were done in the newer formats, I added support
for
S3M's and more recently, XM mods.

Josmod didn't take very much time to write at all, for one reason in
particular: it's written in C. Compared to writing an application in
assembly,
C is so much easier to debug and test. That's one of the great things about
the 65816, you can write in a high level language and get away with it. One
part of Josmod had to be written in 65816, and that's the mixer, which has
to
be optimized as it's where 90% of Josmod's time is spent.

Enough propaganda! On with the show..

-----------------------------------------

There are 3 main parts to any modplayer: the loader, the player routine &
the
mixer.

First of all I will start by talking about the mixer as it's the simplest
part
```

and also the part that deals with the actual sounds themselves.

The Mixer

First of all, I'd like to point out that I'm no expert on digital sound, and I
only learned all this stuff myself when I started writing Josmod. Here's an
ASCII drawing of a digitally sampled sound:



Yes I know it looks crap, but you get the idea. You'll note that it's a SIGNED
sample, which is very important when it comes to mixing.

The Amiga has 4 digital sound channels, each capable of playing its own set of
sampled data, so it can effective play four different sounds at the same time.
But what about the C64? SID's only capable of playing one digi at a time. How
on earth can we get it to play 2 samples at once, or 4? 8 or 16? C64's aren't
alone in their single channel output problem, most PC sound cards only have 1
channel for output (2 if you count Stereo). So how is it done? It's extremely
simple actually. You just add them together! But remember it's signed addition!

The one thing that you have to worry about when adding though, is that you
don't overflow your result. For example, if you're adding two 8 bit numbers
together you could easily go higher than 127 or lower than -128, just by
having two loud samples. So it's best to use a 16 bit result, and handily the
65816 has 16-bit registers. :) On a side note, if a sample is mixed with an
exact opposite of itself, it turns to silence! I believe this is actually used
in some new cars to cut down on noise inside the car! [Editor's note: it is

```
also used in things like aviation headsets, and goes under terms like
"Active
Noise Reduction" and "Active Noise Cancellation"]

Not only does the mixer have to mix samples together, it also has to control
a
couple of other things, namely volume and pitch.  The player reads the notes
and effects, and tells the mixer which samples to play, at what speed and
how
loud they are supposed to be.

I'll start with volume, as it's quite an easy transformation. In Amiga MODs
volumes range from 0 (Silence) to 64 (Full volume). So a volume of 32 would
be
half volume, and effectively divide the samples by 2. e.g. A sample of -40
would become -20 at volume 32, and a sample of 100 would become 50. The best
and fastest way to perform volume calculations in real time is of course
using a lookup table. So you end up with mixer code that looks like this:
(65816 remember!)


        lda [Samp]      ; Get the current sample
        and #$ff        ; It's 8 bits only
        asl             ; Multiply by 2 since the volume
        tay             ; table is 16 bit values
        lda [VolTab],y  ; Get the value!


VolTab contains a pointer to the appropriate volume table, which is
calculated
with:

Sample*Volume/64

Volume was easy but what about pitch? Ordinarily if we were dealing with
normal 1 channel digis we'd simply change our CIA timer rate so data would
be fetched at a different speed, which is ideal. This is basically what the
Amiga does, except it's done in hardware, so doesn't steal valuable CPU
cycles. However, we can't do this as we're mixing various samples into one!
So
instead we have to simulate different pitches by altering the sample
position
at varying rates, rather than just going straight to the next sample.

So for example, if a sample needs to be played at twice its normal speed, we
add 2 to the sample pointer rather than 1, and it will sound twice as
fast. It's all very well adding values like 1 and 2, but what if the sample
needs to be 1.5 times the normal speed? It's easy solved with some fixed
point
math (16 bits for the fraction, 16 bits for the integer part), and you end
up
```

with something like this (remember that Samp is a pointer into the sample data):

```
        txa             ; Add the fraction part
        adc Low         ; X is used to hold the low part
        tax
        lda Samp        ; Now add the integer with carry
        adc Hi
        sta Samp
        bcc noinc       ; Crossed a bank boundary?
        inc Samp+2
noinc   ....
```

Very simple really isn't it?

Note that the tradeoff here is between playback rate and sample rate; that is,
if we have a sound sampled at 10000 samples per second, and take every other sample, that's like taking 5000 samples per second. So instead of changing the _playback_ rate -- which is what e.g. changing the CIA timers does -- we instead effectively change the original _sample_ rate. Another thing worth mentioning is that when the playback rate is greater than the sample rate, you
can improve output by "interpolating" the sample. Interpolating is basically guessing what the sample would have sounded like if it were sampled at the playback rate. I won't go into it any more than that, as adding it will slow the mixer down. I believe Nate's latest modplay64 does interpolation.

That's basically all that the mixer does, but I've left out one part, which is
the part that adds the samples together. Rather than adding 4 values together
at a time the Josmod mixer uses a buffer to add all the values for 1 channel in one go, and then adds all the samples for the next channel to that buffer,
etc. This means that the mixer can handle as many channels as it wants, rather
than being fixed to 4 channels. Here is the code for mixing 2 samples: (Assume
16 bit registers!)

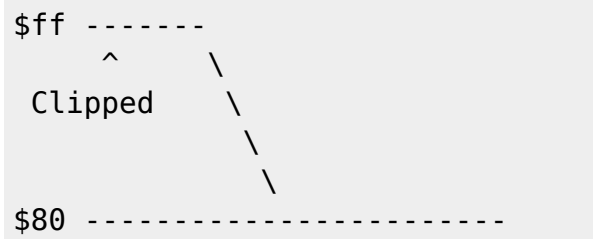
```
        lda [Samp]
        and #$ff
        asl
        tay
        lda [VolTab],y
        adc (OutBuf)            ; Add it to the buffer
        sta (OutBuf)
        txa
        adc Low
        tax
```

```
        lda Samp
        adc Hi
        sta Samp
        bcc ninc
        inc Samp+2
ninc    lda [Samp]
        and #$ff
        asl
        tay
        lda [VolTab],y
        ldy #2                  ; Next buffer position
        adc (OutBuf),y
        sta (OutBuf),y
        txa
        adc Low
        tax
        lda Samp
        adc Hi
        sta Samp
        bcc ninc2
        inc Samp+2
ninc2   ...
```

By changing Samp (and Low/Hi) the same code can add different samples to the buffer. Josmod uses an unrolled mixing loop which does 16 samples before looping, otherwise there'd be too much loop overhead.

You will note that at the end of mixing all the channels what we're left with
is a buffer that is full of 16 bit samples. Well, I believe the IDE64 team have a made a 16 bit sound card... but generally this buffer is not going to be of any use to us, until we convert it to 8 bit unsigned data. The SID uses
4 bit unsigned samples, and the DigiMax and other user port sound cards, use 8
bit unsigned, so it's the best format to convert it to.

Not only does it need to be converted to 8 bit unsigned, but remember the overflow problem? Values that are too loud need to be clipped to their maximum
volume (127 or -128). So we can kill two birds with one stone here, and create
a post processing lookup table that converts the data and also clips loud samples. If it were graphed the table would look something like this:



```
                \
                 \
                  \
                   \
0                   --------
                       ^
                    Clipped
```

So after all channels are mixed together, the buffer is postprocessed with that table to prepare it for output on the sound device! Which incidentally could be any device, as Josmod is device independent, it's up to the device driver to deal with getting a CIA interrupt and what to do with the samples. In the case of 4-bit SID this would be:

```
        lda [Samp]
        lsr
        lsr
        lsr
        lsr
        sta @$d418      ; @ means long addressing - $00d418 SID Volume
register!
```

That's about all for the mixer, the only other thing that it needs to do is to
check if a sample has ended, and either stop it, or loop it back to some point.

Bear in mind that the mixer I've described is a mono one, as all channels get
mixed into the one output buffer. A stereo one would mix channels into two different output buffers.

I should mention that since I wrote this article, I've made some improvements
to the mixer. Namely pre-calculating the sample pointer offset, so that the mixer loop doesn't need to calculate the next sample position for every sample. So now the sample mixing code is much quicker:

```
        lda [Samp]
        and #$ff
        asl
        tay
        lda [VolTab],y
        adc (OutBuf)            ; Add it to the buffer
        sta (OutBuf)
    ldy #1            ; The offset to the next sample
    lda [Samp],y        ; (precalculated)
        and #$ff
        asl
        tay
        lda [VolTab],y
```

```
        ldy #2                  ; Next buffer position
        adc (OutBuf),y
        sta (OutBuf),y
    ....
        txa         ; Add sample speed * 16
        adc Low         ; This code now only gets
        tax         ; executed once per 16
        lda Samp        ; samples rather than every
        adc Hi          ; sample.
        sta Samp
        bcc ninc2
        inc Samp+2
ninc2   ...
```

These changes provide, at a guess, a dramatic increase of about 30 percent in
playback speed. The only drawback to this method is very slight loss of accuracy, but I don't think you will be able to tell the difference.

The Loader

You've now seen how to mix samples, so what you need to know next is how to load in the samples from the modfile and also load the data that tells you how
to play them. Ok let's check out the .mod file format. Sizes are all in bytes.

```
Offs : Size
------------
0    : 20   : Module name
20   : 930  : Sample headers for 31 Samples
950  : 1    : Number of orders
951  : 1    : Unused?
952  : 128  : The orders
1080 : 4    : Identification string
1084 : ?    : The patterns
?    : ?    : The 8 bit sample data for all samples
```

Ok first of all you need to know exactly what orders & patterns are.

Patterns contain all the information about what pitch and sample to play and also contain information about effects which change the volume and pitch of samples. Every pattern contains 64 rows of data. Each row holds note information for the number of channels contained in the tune (normally 4 in Mods). A pattern can be visualised as something like this:

```
        1       2       3       4
-------------------------------------
1:  | 1 c-1 | 2 d-2 | 2 f#5 | 3 e-4 |
2:  | 5 d-1 | - --- | - --- | 3 f-4 |
```

```
......
64: | 4 e-1 | 3 d-1 | - --- | - --- |
-------------------------------------
```

That's a simplified version of what a pattern is like, as it doesn't include any effect information.

So what does "1 c-1" mean?

Well the first 1 refers to the sample number, while "c-1" refers to the note and octave to play (C natural, octave 1).

Orders will be familiar to anyone who's had any experience with some of the SID editors/players. Rather than playing pattern 0, pattern 1, etc.. through to the last pattern, orders allow you to choose the order in which patterns are played, thus allowing you to repeat patterns easily. E.g. The order table
may look something like this: 0 1 2 3 2 2 4 3 5 6

You will notice that nowhere in the mod format does it mention how many patterns are contained in the file.. To calculate this you look through the order table and find the highest value, and use that as the number of patterns
in the file (+1).

Ok now let's have a look at the sample header format:

```
0    : 22   : Name
22   : 2    : Length (divided by 2)
24   : 1    : Finetune
25   : 1    : Default Volume (0-64)
26   : 2    : Loop position (divided by 2)
28   : 2    : Loop length (divided by 2)
30
```

One thing to be warey of here is the fact that the 16 bit values stored in this structure are in big endian format, because Amigas use the 680x0 line of
processors. So to convert it for use with C64 it needs the high and low bytes
swapped around. Also I have no idea why the 16 bit values are stored divided by 2... I guess this is something to do with playing the mod on the Amiga too,
or perhaps to allow larger than 64k samples?

Ok now we know what the format of a .mod file is, it's time to get it loaded so our player can play it!

You will notice that the first 1084 bytes of the file are fixed, so one approach to loading would be to load those first 1084 bytes and go on from there, but the approach I use is to load each part of the header seperately,

which makes it easier to put the data in a format easier for the player. Ok let's take a look at the C source for Josmod's .mod loader..

First a couple of declaration for identifying which mod format it is:

```
typedef struct IdentStruct {
    char *String;
    int channels;
} Ident;

static Ident idents[] = {
        {"2CHN", 2},
        {"M.K.", 4},
        {"M!K!", 4},
        {"FLT4", 4},
        {"4CHN", 4},
        {"6CHN", 6},
        {"8CHN", 8},
        {"CD81", 8}
};

int loadMod(char *name, ModHead *mp) {

    /* Variable declarations */

    FILE *fp;
    S3MSamp *samp;
    unsigned char *patp;
    unsigned int i,j,temp,numpat,temp2;
    long patsize;
    char *samdata;
    Ident *idp;
    /* Open the mod */
    fp = fopen(name, "rb");
    if (!fp) {
        perror("josmod");
        exit(1);
    }
    /* Read the name of the mod */
    fread(mp->Name,1,20,fp);
    mp->Name[20]=0;        // C strings are null terminated!

    /* Read 31 sample headers */
    samp = &mp->Samples[0];
    for (i=0;i<31;i++) {
        /* Read name and null terminate */
        fread(samp->Name,1,22,fp);
        samp->Name[22]=0;
        /* Read length
        Note: readWord() converts the 16 bit value
        into C64 format
```

```
        */
        samp->Length = readWord(fp) << 1;
        /* Read and calculate finetune
        Finetune is a 4 bit signed value
        Values over 8 are negative values
        */
        samp->Finetune = fgetc(fp);
        if (samp->Finetune>=8)
            samp->Finetune -= 16;
        /* Read volume */
        samp->Volume = fgetc(fp);
        /* Read and calculate ending/looping positions */
        temp = readWord(fp) << 1;
        samp->Replen = readWord(fp) << 1;
        /* If the loop length is higher than 2, this
        is a looping sample, otherwise it's a standard
        non looping sample that stops at the end */
        if (samp->Replen > 2) {
            samp->Looped = 1;
            samp->End = (char *) temp + samp->Replen;
            if ((unsigned int) samp->End > samp->Length)
                samp->End = (char *) samp->Length;
        } else {
            samp->End = (char *) samp->Length;
            samp->Looped = 0;
        }
        /* Go to next sample structure */
        samp++;
    }
    /* Read orders and calculate number of patterns */
    mp->NumOrders = fgetc(fp); fgetc(fp);
    numpat=0;
    for (i=0;i<128;i++) {
         temp = fgetc(fp);
         if (temp > numpat)
             numpat = temp;
         mp->Orders[i] = temp;
    }
    numpat++;
    mp->NumPatterns = numpat;

    /* Identify type of mod */
    fread(mp->Type,1,4,fp);
    mp->Type[4]=0;

    mp->Channels=0;
    idp = &idents[0];
    for (i=0;i<8;i++) {
        if (!strncmp(idp->String,mp->Type,4)) {
            mp->Channels=idp->channels;
            continue;
```

```
        }
        idp++;
    }
    /* If we couldn't identify it, this isn't a mod file!
    close the file and return. Otherwise print number of
    channels. */
    if (!mp->Channels) {
        fclose(fp);
        return 0;
    } else {
        printf("Channels %d\n",mp->Channels);
    }


    /* Calculate pattern sizes and allocate memory for patterns */
    mp->LineSize = 4 * mp->Channels;
    mp->PatSize = 64 * mp->LineSize;
    patsize = mp->PatSize * numpat;
    patp = xmalloc(patsize);
    mp->Patterns = patp;
    /* Load patterns */

    for (i=0;i<numpat;i++) {
        for (j=mp->PatSize/4;j;j--) {
            /* Each note is stored in 4 bytes.
            Contained in the 4 bytes is:
            Sample number.
            Note
            Effect
            Effect Parameter
            conv2Note() converts the mod format note format
            into a format easier for Josmod.
            */
            * (unsigned int *) (patp+2) = readWord(fp);
            * (unsigned int *) patp = readWord(fp);
            conv2Note(patp);
            patp += 4;
        }
    }

    /* Load Samples */

    samp = &mp->Samples[0];
    for (i=0;i<31;i++) {
        printf("Sample %d: %s\n",i,samp->Name);
        if (samp->Length) {
            /* If the sample exists, allocate memory
            for it, and read it in. Then fix up it's end
            pointer, then call fixSamp() to prepare it for
            mixer output. */
            samdata = xmalloc((long) samp->Length + 128);
```

```
            fread(samdata,1,samp->Length,fp);
            samp->Samp = samdata;
            samp->End = (unsigned long) samp->End + samdata;
            fixSamp(samp->Looped, samp->End, samp->Replen, 128);
        } else samp->Samp = NULL;
        samp++;
    }
    /* Close the file and return successfully */
    fclose(fp);
    return  1;
}
```

That's all for the loader! Now that patterns, orders, and samples are all loaded and in a suitable format, the player is ready to play it!

Unfortunately that's all for now, as the player is quite a complicated topic,
I'll save it for a later article. But before I finish, I do want to give you a
basic outline of what happens in the player, so you can see where the mixer fits into all this.

The Player

Mods have two speed variables which control the way a mod is played. One is BPM, or Beats Per Minute. BPM specifies the speed at which the mod player routine is called, the default value is 125 BPM, or 50 times a second. As we all know, 50hz is the speed of PAL TV's, so it's no coincidence that this is the default, as it allowed Amiga players to setup their play routines on an interrupt synchronised with the screen, the same as with SID players.

BUT since we don't have the luxury of having hardware to play the samples for
us, it's not a good idea to hook our player to a raster interrupt, instead we
calculate how many output samples need to mixed before we update our mod. We calculate that with the following:

playbackrate / (BPM*2 / 5)

This value is what I call the "TickSize". Everytime the player is updated, it
will mix TickSize bytes of samples, then loop and update the mod again, mix TickSize bytes again, etc..

The other speed variable is the number of Ticks per beat, and defaults to 6. This means that the mod player only fetches the next row of pattern data every
6 Ticks. E.g. After 6*TickSize samples have been mixed.

```
Each channel in the Mod has assocatied with it, a mixer structure, which
looks
like this:

typedef struct {
    char *Samp;        // Current sample position
    char *End;        // End or loop position
    unsigned long Replen;    // How far to loop back
    unsigned long Speed;    // Speed/pitch of sample
    int Low;        // Fractional part of sample position
    int Repeats;        // Whether or not it repeats
    int *VolTab;        // Pointer to volume table
    int Active;        // Whether to mix this channel
                // or not
} MixChan;

The player has to prepare these structures for each channel and on every
tick
update the appropriate fields depending on the data contained in the
patterns.

I'll leave you with psuedo code for what the player looks like:

tickbeat = 6
row = 64
tickcount = 0
orderup = 0
make all channels inactive
while (orderup < numorders) {
    if (row == 64) {
        get next pattern from orders
        orderup++
    }
    if (tickcount == 0) {
        get row data from pattern
        process note - set sample speed
        process sample number - set sample position volume
        process effect - speed, volume and position effects
    } else {
        process effect
    }
    mix ticksize samples
    if (mixing buffer full) {
        play buffer
    }
    tickcount++
    if (tickcount >= tickbeat) {
        tickcount = 0
        move to next row
        row++
    }
```

```
}

That's basically what the mixer does. The majority of the player code deals
with effects processing, most of which are easy to process. Effects include:

Portamento - sliding a note's pitch up or down.
Volume setting/sliding
BPM and Tick Beat changes
Pattern looping
Vibrato and Tremolo - Slight alterations in pitch and volume.

Some effects are only processed on Tick 0, and some are processed on other
ticks.

Anyway that's all for now, the player code will be examined in a later
article.  In the meantime you can view the source code in HTML form at:
http://jos64.com/src/josmod.c.html
.......
....
..
.                                     C=H #20
```

The C64 Digi

```
Robin Harbron <macbeth@psw.ca>
 Levente Harsfalvi <levente@terrasoft.hu>
  Stephen Judd <sjudd@ffd2.com>

Introduction
------------

Digis -- digitally sampled audio -- are fairly common on the 64.  This is
meant to be a comprehensive article on digis: how they work, examples,
different playback methods on the 64 (volume register and Pulse Width
Modulation), and some tricks.  We'll even show you how to play 6-bit and
even 8-bit digis in high quality on a 64, which is really pretty neat to
hear.

The first part discusses digis from a fundamental point of view -- just
what a digi is, acoustic signals, and things like that.  The most common
method of playing digis is via the volume register at $d418, and the next
two sections are devoted to this technique.  Section two discusses some
SID fundamentals, and the reason why $d418 may be used for digis (and why
later-model SIDs don't play digis correctly); Section three discusses
$d418-digis from a software perspective: how to play them, tricks for
improving them, how to boost digis on 8580 SIDs, and how to detect what
kind of SID (6581 or 8580) is in the machine.  The fourth and final part
of this article discusses pulse width modulation, and includes example
source
```

code and a binary that plays a true 7-bit digi at around 16KHz -- something which, we think, has never been done before.

Without further ado...

==========
Digis: Overview
==========

The whole point of playing a digi on a 64 is to provide something for your ear to hear. So let's begin by discussing just what an acoustic signal is and how that relates to digis.

Probably everyone knows that "sound" is how your ear responds to changes in air pressure -- that is, when you clap your hands together, it compresses the air between your hands in a special way, and that higher pressure moves outwards into the surrounding air (since it's at lower pressure). That pressure change propagates along and when it encounters your ear it causes the ear drums to move, causing three little bones to move, causing some fluid to move, causing tiny, exquisitely sensitive hairs to move, transmitting a signal that your brain converts to "sound".

An audio speaker also changes the air pressure in response to a signal. If you take a coil of wire and change the voltage on it, it generates a magnetic field; if a magnet is placed inside the coil, the changing magnetic field will place a force on the magnet, causing it to move, causing some air to be pushed along, causing a change in pressure, causing a signal to propagate to your ear which your brain interprets as Van Halen. All a stereo (CD player, etc.) does is send a varying voltage signal to the speaker. As that voltage level goes up and down the magnet moves back and forth, and so the speaker converts that electrical energy into an accoustic wave.

For us, the trick is to coax SID into sending a specific voltage signal to the speaker, the way a stereo or CD player might. And a CD player is of course a very apt comparison, since it is itself a digi player.

Just for reference, a really good pair of ears can hear signals from around 20Hz to 22KHz, with the sensitivity dropping considerably outside of around 100Hz to 10KHz. A CD player has a playback rate of 44KHz, and the highest frequency SID can generate from the frequency registers is around 4KHz. If you've ever set SID to maximum frequency and heard just how high 4KHz is, you can appreciate that even 10KHz is _really_ high, and actually quite difficult to hear. In human speech, most of the information content of vowel sounds is contained in the range 300Hz - 3KHz, and above around 1KHz for consonant sounds; most information in musical sounds is in the range 100Hz - 3KHz.

Discrete Sampling

```
    To understand digis a little better, consider the more general
case of a discretely sampled signal -- a continuous signal sampled at
discrete time intervals.  Let's say we had some device producing a
_continuous_ sinusoidal signal in time:

          *  *
       *         *
     *             *
    *               *
   *                 *
  *                   *                        *
*                      *                      *
                        *                    *
                          *                *
                            *            *
                              *        *
                                *    *
                                  *  *
-------------------------------------------------> time

(yes, I did miss my calling as an ASCII artist)

To turn the signal into a _discrete_ signal, we simply sample the
signal at discrete intervals of time.  For example, let's say the above
signal lasts one second, and is input into a device which measures the
value every 1/4-second.  The device will spit out four numbers: 0, 1, 0,
and -1:

           *



*                    *



                                 *

The sampling frequency here is four samples per second -- 4 Hz.  If we were
to then play back this signal at the sampling frequency, we'd get a signal
like

           **********


**********           **********



                                 **********

So one thing sampling does is to "staircase" a signal -- the sample becomes
some sort of "average" value over the sample period.  Increasing the sample
rate -- taking more samples per second -- will smooth things out, and the
```

```
sampled signal will look (and sound!) more like the original signal.

Now let's say we just took two samples in that one second -- 2 Hz sampling
rate -- and just happened to catch the signal at its maximum and minimum
values (the peak and trough).  Upon playback, the signal would look like

**********************
                     *
                     *
                     *
                     *
                     *
                     *
            **********************

That is, a square (pulse) wave.  If you're on the ball, you've noticed
that the frequency of the new signal is 1 Hz -- exactly half the sampling
frequency.  This is also called the Nyquist frequency.  In general, the
_maximum_ frequency that can be captured in a discrete sample (called the
Nyquist critical frequency) is half the sampling frequency -- as you can
see above, it takes two data points to get a single (nonzero) frequency.
So, for example, the highest frequency a CD player -- which has a sampling/
playback rate of 44KHz -- can capture is 22KHz, well above the range of
normal human hearing.

Thus, increasing the sample rate increases the frequency range captured
in the discrete signal.  This is why a digi at a high sample rate in general
sounds better than a digi sampled at a low sample rate.

BUT -- there is more to life than sample rate: there is also sample
resolution.  The sample resolution -- 4-bit samples, 8-bit samples, etc. --
determines how accurately the sample measures the actual signal.  For
example, let's say we sample sin(x) when x=0.5:

    sin(0.5) = 0.4794255...

No matter what sample resolution we use, there will always be some error
in the measurement, and the _true_ value of the sample will be the
_measured_ value plus some error.

In general the sampling errors are random and uniformly distributed, so
the sampled signal corresponds to the original signal plus some noise (the
random errors).  That is why you almost always hear some sort of hiss on
a normal C64 digi, which uses a resolution of 4 bits per sample.

So, increasing the sample _resolution_ decreases the amount of noise
introduced
into the sampled signal (and increases the dynamic range), and increasing
the
sample _rate_ increases the frequency range.
```

If you're _really_ on the ball, you've noticed that the 1-Hz square pulse above actually contains frequencies higher than 1Hz, simply because a square pulse contains higher harmonics in addition to the 1Hz fundamental frequency. And you've also no doubt realized that the sampled pulse wave would sound different than the original sine wave (due, of course, to the added harmonics) -- it's at the right frequency, but it will sound like a pulse wave instead of a sinusoid.

Have we somehow broken the Nyquist limit?

The answer is no, because of a nifty thing called the Discrete Sampling Theorem, which says that, given the samples h_n of a bandwidth-limited function h(t), the original function h(t) is given by

    h(t) = dt * Sum{ h_n * sin(2*pi*f_c*(t-n*dt)) / (pi*(t-n*dt)) }

where dt is the sampling period and f_c is the cutoff/critical frequency.

What this means is that the original signal can be _reconstructed_ from the discrete samples, not that it is _equivalent_ to the discrete samples.
The Nyquist limit is the highest frequency that can be _reconstructed_ from the discrete samples, not the highest frequency that will be produced if you "staircase" the discrete samples through a speaker. If the original signal is bandwidth-limited, and there are at least two samples for the highest frequency, then the signal can be completely reconstructed.

Since a "normal" digi contains all these extra frequencies, shouldn't a digi sound "different" than a "true" analog signal? Sure. On the other hand, many
of the extra frequencies are beyond the range of human hearing, and the rest can often be removed using a filter -- all CD players filter the output, for example. So sometimes it is worthwhile to turn on a low/band pass filter when playing a C64 digi, especially at lower sample rates.

And that more or less summarizes basic discrete sampling theory.

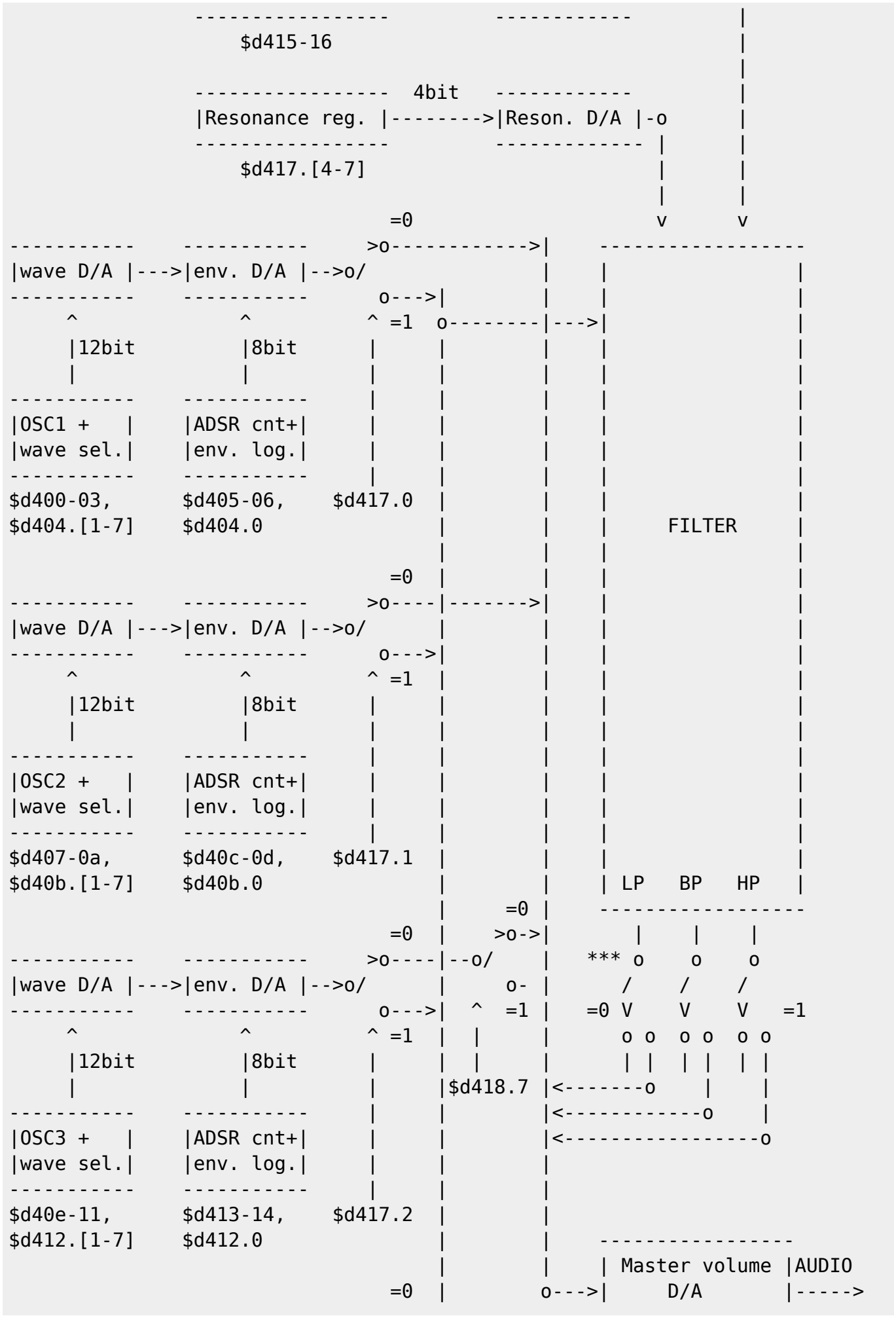
==========
D418 Playback -- Hardware
==========

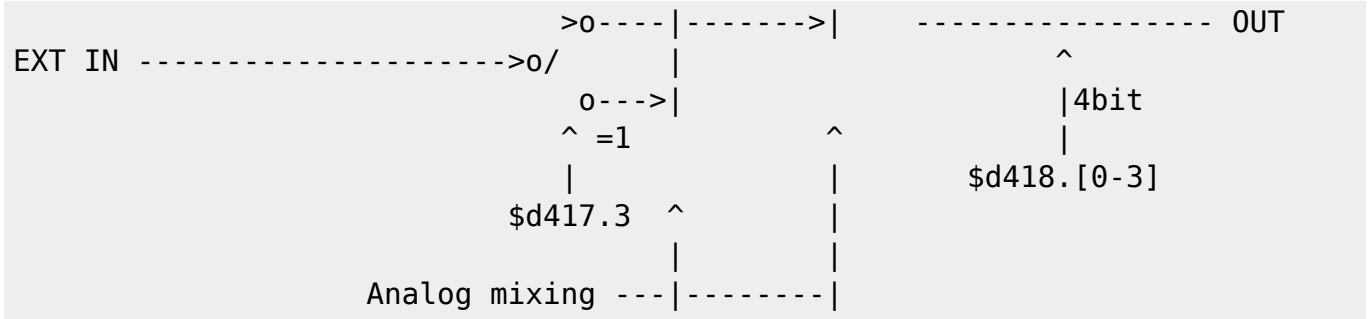
The SID contains both analog and digital subparts on one silicon plate -- in other words, it is a mixed signal device.

At the time, the SID was certainly the best of the microcomputer sound chips.
This may be mostly due to its mixed signal design, which the designers used to solve certain problems.

The hard thing in a sound generator design is to implement waveforms, volume control, and mixing. Things like that don't really fit into the digital

```
'either 0 or 1' philosophy, unless lot of data bits and arithmetic functions
are involved. In a fully digital sound chip, the waveforms could be
generated
by ROM lookup tables. The mixing function could be derived from binary
addition, while the volume control from division or multiplication. Unless
the
sound functionality is greatly simplified, the arithmetic functions must be
present and they must be implemented in hardware. Finally, the D/A
conversion
could be done by (fast) pulse width modulation just at the output stage.
(Most of today's wavetable sound cards operate like this).

This method implies heavy arithmetic hardware, which was not an option for
designers back then. Still, most sound chips were fully digital, and all
suffer from the required compromises (i.e. generating square waves only,
no dedicated channel volume control, etc. - both TED and the VIC-I are
obvious
examples).

The solution that one finds in the SID design is very straightforward:
mixing
and variable volume level is problematic in a digital circuit when dealing
with waveforms, so simply avoid doing it. In the SID, only the microcomputer
interface, the registers, the oscillators (phase accumulating oscillators),
and other controller logic are digital; the mixing and volume control parts
are fully analog. There are digital to analog converters providing analog
voltage levels from the digital state variables. The SID D/As are in fact
'multiplying' D/As, having an analog input (AIN), an input base voltage
(IBASE), and a digital input. They operate by amplifying the input voltage
offset (AIN-IBASE) by a factor proportional to the number on the digital
input
and adding this offset back to the base level.

This mixed signal design also allowed some other features to be implemented.
The most important one is the analog filter (that is, a two integrator loop,
bi-quadratic filter, according to Yannes). With that, the SID points beyond
a
home computer sound chip - it is a true analog subtractive synth (marketing
as
such was cancelled because of manufacturing capacity reasons).

Here is a detailed map on the SID inners (analog path; probably my most
beautiful ASCII ever :-D). Info can be found in the SID patents (US
4,677,890;
1986), the MOS 6581 technical document (can be found somewhere on the Net),
or
the back of the Programmer's Reference Guide (PRG).


                ----------------  11bit  ------------
                |Cutoff freq reg|-------->|Cutoff D/A|---------o
```


$d415-16
4bit
Resonance reg.
Reson. D/A
$d417.[4-7]
wave D/A
env. D/A
OSC1 + wave sel.
ADSR cnt+ env. log.
$d400-03, $d404.[1-7]
$d405-06, $d404.0
$d417.0
12bit
8bit
=1
=0
FILTER
OSC2 + wave sel.
$d407-0a, $d40b.[1-7]
$d40c-0d, $d40b.0
$d417.1
LP BP HP
OSC3 + wave sel.
$d40e-11, $d412.[1-7]
$d413-14, $d412.0
$d417.2
$d418.7
Master volume
AUDIO D/A



***: Filter type select switches, $d418.[4-6] respectively

$d418 digis

The most common method of playing a digi is to use the register at $d418. When someone plays a digi using the master volume register, the situation is similar to the waveform D/A converters. Both D/As are multiplying D/As -- signal amplifiers whose amplification is proportional to the input digital number. If there is a nonzero signal offset on the D/A input it will be multiplied proportionally by this number.

Playing digis with $d418 is possible because there is indeed a relatively large DC voltage offset on the master volume D/A. This offset is present right from the moment when the SID is powered up.

Where can this DC offset come from?

There is a mixer before the master volume D/A (see figure). If there's a DC offset on the D/A input, it must come from there. ...And going further, the DC offset on the mixer must also come from somewhere. But where?

Signals come from the three ADSR volume D/As, the EXTIN line, and the three outputs of the filter. Fortunately, all paths that go to the mixer have analog
switches (all paths can be disconnected from the mixer individually, if that's
needed).

The above analog switches are driven by the filter selection bits ($d417 bits
0-3), the voice 3 off bit ($d418 bit 7) and the filter type selection bits ($d418 bits 4-6).

After a reset, the filter selector bits are all 0 (all signals are routed towards the master mixer), the 'voice 3 off' switch is on, and the filter type
selector bits are 0 (filter outputs are unconnected). In this state, only EXTIN and the three SID voice signals are present on the mixer. EXTIN can be eliminated as the source since it has no DC offset (as long as the computer

```
was not hacked, see notes on the 8580).

The ADSR volume D/A is similar to the previously mentioned multiplying D/As.
If the digital number on the input is 0, the input analog signal offset
can't
pass through (as measurements verify). This is the case when SID is reset,
setting the envelope counters to zero.  Therefore, nothing behind the ADSR
multiplying D/As can have any effect on the DC offset of the mixer.

So, the DC offset must come from the ADSR multiplying D/As. Another
measurement shows that even the mixer itself has a small DC offset.


Tests and results
-----------------

I did some tests that support this theory. They were done 'by hand', by
simply
using a digital voltmeter + the FC3 monitor.

The chip was a 6581(R1), 0883, Hong Kong (an early 6581).

When turned on the voltage on the AUDIO OUT was about 5.5 volts (slowly
decreasing as it warmed up, stopping at about 5.43 after some 10 mins - all
subsequent tests were done after this time period).

Writing $0f to $d418 raised the output voltage to 6.15 volts. Therefore, the
maximum output amplitude that can be achieved when playing digis is 0.72
volts
in this 'mode' (without wiggling any other SID settings to achieve higher
voltage levels) -- remember that what counts is the maximum voltage
_difference_, not the maximum absolute voltage.

The next test is to determine if the mixer has its own DC offset (with all
possible paths are disconnected). It's possible to do. With the volume at
maximum (to maximize any effect), all voices are routed towards the filter
($d417 = $0f), while making sure that the filter outputs are not routed to
the
mixer ($d418 = $0f). In this state no paths can drive the mixer. The result
is 5.39 volts. When the volume changes, the output also changes towards the
previous 5.43 volts --> there is a (very small) DC offset from just the
mixer.

What could be the DC offset value of each individual SID voice (i.e. the
base
level difference of the multiplying D/As)?  Doing the above, but leaving one
voice routed to the mixer ($d417 = $0e, $0d or $0b) gives 5.69 volts.
5.69-5.43 = 0.26 volts, and 5.43 + 3*0.26 = 6.21, almost 6.15 volts.

To determine if the ADSR multiplying D/As act as expected, I used pulse
waves with zero frequency and 0 or $fff pulse width (two cases), to make the
```

input signal of the ADSR multiplying D/A the minimum and maximum possible level. After careful checking, the output changed a few hundredth volts (about 0.01 volt per voice). So the D/A doesn't close up completely, but it's still O.K.

To prove that these offsets are equal for all voices, I did another test. Some
people know that the filter inverts phase (multiplies the input signal by -1).
Machine is reset, $d417 = $01, $d418 = $9f. (Voice 1 is routed through the filter, voice 3 is cut off from the mixer completely ($d418.7), low pass filter is selected, volume = $0f). The output voltage was 5.41 volts, just very slightly below the "default" output level. This means that the DC of voice2 + (-1*) DC of voice 1 resulted in about 0 relative offset. Doing similar tests proved that the DC offsets for the voices match each other almost exactly (within a few hundredths of a volt).

These measurements all support the idea that the DC offset comes from the ADSR multiplying D/As, that the offset is mostly independent from the waveform
D/A converters (as long as sustain levels are 0), and that the offsets are equal for all voices. In addition, a small DC offset is supplied by the master
signal mixer itself.

What if we try different sustain settings? For this test, set the volume to maximum, as usual. Set the sustain level to $0f for all voices ($d406, $d40d,
$d414 = $f0). Start the attack, but with no waveform selected ($d404, $d40b, $d412 = 01). The output level is now 5.21 volts, a little bit below the '0' offset of the audio output! (Doing the test with just one voice (all others disconnected), the output is 5.29 volts).

Finally, we can do some experiments with the pulse waveform. The pulse waveform is useful for these tests, since at zero frequency we can set both the minimum and the maximum constant DC levels at the voice D/A just by using
the pulse width registers. Reset the computer. Set voice 1 to zero frequency,
pulse level $0fff, sustain level 15, and $d404=$41 (pulse waveform + gate on).
Route only voice 1 through the mixer ($d417 = $0e). The output voltage is similar to the test when no waveform was selected -- 5.29 volts! This seems to
show that "waveform accu = $0fff" is the same as when no waveform is selected
(i.e. the waveform D/A digital input pins are pulled high when they're not driven, as seen in most other NMOS chips).

When the pulse width is 0 in the above test the output changes to 6.34 volts.

This seems to be strange (a multiplying D/A giving higher signal level for multiplying something by 0).

Now, when the ADSR multiplying D/A is closed, the output is 5.70 volts. When it's fully open, the output changes from 5.29 volts (wave acc= $fff) to 6.34 volts (wave acc = 0). One reasonable answer is that the base voltage of the waveform D/A is higher, and the analog input is tied lower than the base voltage of the ADSR D/A -- the effect is that the SID waveforms will lie 'around' the ADSR multiplying D/A base voltage, more or less symmetrically.

This was surely done intentionally, to reduce absolute voltage levels (for linearity). In the 6581, the big DC offset is probably a result of having the
ADSR D/As and the master volume D/A at different base levels (the difference appears as true DC offset on the master volume D/A). If both were the same (presumably at VDD/2), and the waveform D/A parameters were selected similarly
(operation is symmetric to VDD/2), there would be no final DC offset at all. Rather like the 8580...

Other issues

So now we know why $d418 digis are possible - but still, there are some things
to note.

The DC offset on the master volume D/A changes with different SID settings, and whatever affects the DC offset on the mixer will affect the digi volume. For example, even the filter output signals have a small DC offset. Just do a test - set the volume to 0f, then simply turn one filter route on (for example, $d418 = $1f). You'll hear a small click (i.e. a small DC offset
change on the mixer), even if the filter has no input.

Moreover, as seen above, the DC offset can be eliminated completely (just by SID register settings), leading to no audible digi sound at the output. In other words, whatever affects the DC offset on the mixer _will_ affect the digi volume.

One place where this is important is playing a digi with a tune: there's a constantly changing signal going to the mixer instead of a constant DC offset,
so playing a digi on the master volume also causes distortion for both the SID
voices and the digi sound (since they're cross modulated). To reduce this effect most 3+1 like SID + digi players play samples by writing 8-offset sample
values to $d418 (ie. adding 8 to 3-bit sample values and writing this to $d418

```
- see players used by Jeroen Tel and other famous composers using digi).
This
trick reduces the modulating effect while still maintaining good digi
volume.

The DC offsets used to create awful clicking sometimes. For example, the
filter inverts phase. If the filter is currently routed to the mixer,
there'll
be a large 'click' (2 times the DC offset) when a voice is on and its
routing
is changed to or from the filter.


The 8580
--------

This is a completely redesigned chip. I don't know details, but it was
probably redesigned by the time all other chips in the C64 were done for
CSGs
new manufacturing technology and the C64c. It is a 'better' chip from the
technical side (but in my opinion it sounds crude in comparison to the 6581,
at least the R4 series). The 6581 was designed in months. Bob Yannes had to
do
everything from scratch and use the manufacturing technology MOS currently
had
(NMOS). And it shows.  First, it has high background noise. The DC offsets
are
really also a misfeature. The D/A converters are sometimes non-monotonic (at
least, the waveform D/As and the filter cutoff D/A have some drops at the
change of the most significant bits). The op-amps in the active (resonant)
filter are simple, linearized NMOS inverters ;-) (loopbacked, they act like
more or less linear op-amplifiers around VDD/2). And I still haven't
mentioned
bugs in the digital side (ADSR envelope bugs). Because of the above, one
probably won't find two identical 6581 chips -- each sounds a little bit
different (mostly due to the filter). Since the active components of the
filter are far from ideal, the filter is strongly nonlinear (the cutoff
curve
changes with signal amplitude). On the other hand, these things are what
make
the SID sound so unique.

Most of the problems were fixed in the 8580. It has much less background
noise. The chips sound the same (there are hardly any differences between
different 8580s). Most of the DC offset issues (the clicks) were elminated.
It
needs less power, and lower VDD level. Something was changed also in the
digital logic, but the ADSR part was not touched. The 'combined' waveforms
are
a bit different (and more useable from the musician's point of view).
```

The clicks were reduced, which means that there is no (or no significant) DC offset on the master volume D/A in the 8580.

(I have not done any measurements, but after listening to a lot of 3+1 channel
type musics, I have a strong suspicion that even if sounds are turned on, the
average DC offset on the master volume D/A is still minimal).

To fix this in software, you'll have to wait until the next section of this article.

To fix this in hardware, people use a simple hack: take a resistor of about 330k and tie the SID EXTIN line to GND through that (directly, beside the chip, on the mainboard).

The EXTIN line goes directly to the mixer, and thus the master volume D/A, or
can also be routed through the filter. In either case, unless the filter is disconnected, the above hack will give a pretty large DC offset, similar to the original 6581s. So, digi sounds can be played :-) (even with SID music playing simultaneously, similar to the 6581).

This solution is good as a work around, but there's one thing to note: this is
not completely the same as the 6581 ADSR D/A offset voltage. At least, this offset is negative (should that pin rather be tied to VDD?). Programs that depend on the 6581s way of DC offsets will not work correctly (but I know of very few such programs, so at worst you'll experience slightly different digi
sound only occasionally -- but hey, the 8580 sounds different anyway). Another
problem is that when EXTIN is routed through the filter the DC offset may cause strong distortion since the DC operating point of the filter is changed
-- bad news if the 'semi-linear' amplifiers in the filter are picky about absolute DC level. Some music (not neccessarily involved with digis) indeed do route EXTIN through the filter, for noise reduction on older C64s (with the
earlier C64 mainboards that pick up lots of 'digital' background noise from EXTIN). DC distortion can also occur occasionally for the same reason but on the master volume D/A (the higher the difference from VDD/2, the greater the risk of experiencing nonlinearity and clipping distortion).

Some final words

A lot of this information comes from Dag Lem, who is certainly the No. 1 SID hacker for me ;-). Take a look at reSID, his SID emulator library (the sources

```
can be downloaded from somewhere). reSID contains so much reverse engineered
information of the real SID that you won't believe it -- check it out if
you're interested.


=============
D418 Playback -- Software
=============

$D418 digis are by far the most common playback method.  The volume register
gives 16 different amplitudes (0-15), and so can provide 4-bit digi
playback.

In its most basic form, this is an extremely easy routine to code.  Simply
load each 4-bit sample, and store it in the volume register ($d418).
Assuming $fd/$fe are pointing to the beginning of a series of samples, the
following code will play it back:

     ldy #0

:loop    lda ($fd),y
     sta $d418

     ldx #5 ;some delay value
:delay   dex
     bne :delay

     iny
     bne :loop

     inc $fe
     jmp :loop

The ldx #5 would have to be adjusted depending on the speed of the
sample - the lower this number (not including zero) the faster the
sample will play back.

There are a number of improvements we could make to this code - first
of all, this method takes twice as much RAM to store the sample as is
necessary.  Because we're dealing with 4-bit samples, we can store 2
samples in each byte.  This can be handled simply by alternately
masking out the high bits (with AND #15) to play the sample stored in
the low nybble, and by shifting the high nybble down to the low nybble
to play the high nybble (LSR : LSR : LSR : LSR).  A lookup table may also
be used to save processor cycles (but use more RAM).

Another improvement is to move the routine to zero-page, and use self-
modifying code.  In general, this results in the fastest digi players.

We should of course have the routine check for the end of the sample --
typically just checking the high byte of the zero-page pointer is enough
```

```
(in this case, checking $fe).  Typically digis are page aligned anyway, so
just zeroing out the unused part (if any) of the last page is fine.

Finally, it is often important that each sample of a digi is played back
at regular intervals.  If the samples aren't played at a steady speed,
extra distortion is audible.  In the example above, playback is steady
for a full page (256) of samples - but several extra cycles are added
by incrementing the zero-page pointer to the digi.  The situation
worsens when we start adding extra code to check for the end of the
digi, and even the main loop starts getting irregular when we add the
code for the simple form of packing discussed earlier (2 4-bit samples
per byte).

These problems can be solved by careful cycle counting and adding NOP
and harmless BIT instructions in strategic places to make each
iteration the same number of cycles, regardless of which branch is
taken - people who have written a stable raster routine, or done some
Atari 2600 coding have likely done this sort of painstaking work
before.


NMI-driven digis
----------------

More commonly, however, we enlist the help of CIA #2 and have it
generate regular Non-Maskable Interrupts which we use to call our digi
player.  This has two important advantages - first, it makes timing
much more simple.  Second, it frees your main program to do other
things while the digi is playing "in the background".

To experiment, I pulled a 4-bit packed digi from the extras disk
included with Super Snapshot 5.22.  It's the beginning seconds of the
introduction to Classic Star Trek (Space, the Final Frontier).

Here's the source for a fairly "frills-free" NMI based digi player,
with my comments after blocks of code:

start    = $1400
end      = $7cff
freq     = 141
ptr      = $fd

Labels start and end simply point to the beginning and end of the
digi.  Freq isn't actually the frequency - it's the number of
processor cycles between interrupts necessary to play the digi at the
desired speed/pitch.  If you know the frequency (in hz) of the digi,
simply divide your CIA clock speed (approximately 1000000 hz) by the
digi frequency.  In this case, the digi runs at approximately 7100 hz.

We use two zero page locations to form a 16-bit pointer to the current
sample in the digi to play.
```

```
         *= $1000

         ;disable interrupts
         lda #$7f
         sta $dc0d
         sta $dd0d
         lda $dc0d
         lda $dd0d
         sei
```

This code simply disables interrupts and initializes both CIA timers.

```
         ;blank screen
         lda $d011
         and #255-16
         sta $d011
```

Just like erratically timed code can introduce distortion when a digi is played back, the VIC steals cycles from the processor that can cause interrupts to not occur precisely when you'd like them to. This routine will work without the screen blanked, but the extra noise introduced when the screen is on is noticeable when the time between samples is less than around 2.5-3 times the time the processor is stopped. Another option is to use some multiple of the raster timing as the sampling rate, and start the routine on a non-badline, to ensure that the interrupts never occur on a badline. (A final option is to use a raster-driven interrupt for the digi; with the SCPU, it is actually possible to drive an IFLI display and play a digi at the same time, badlines and all -- email Robin for more info, or maybe wait for a future article!). But the simplest thing to do is to blank the screen :).

```
         ;switch out roms
         lda #$35
         sta 1

         ;point to our player routine
         lda #<nmi
         sta $fffa
         lda #>nmi
         sta $fffb
```

Unless using the KERNAL routines is necessary in my program, I always switch out the ROMs. One of the biggest benefits is that our NMI routine will be immediately called, rather than using $0318/$0319 and waiting for the KERNAL to indirectly call your routine.

```
         ;initialize player
```

```
         lda #<start
         sta ptr
         lda #>start
         sta ptr+1

         ldy #0
         sty flag
         lda (ptr),y
         sta sample

This section simply initializes the various memory locations that the
player uses - sets ptr/ptr+1 to point to the beginning of the digi,
loads the first sample, and clears the flag that handles the
alternating between the lower and upper nybble of the packed samples.

         ;setup CIA #2
         lda #<freq
         sta $dd04
         lda #>freq
         sta $dd05

Sets Timer A on CIA #2 to freq.

         lda #%10000001
         sta $dd0d

Enables Timer A interrupts on CIA #2.

         lda #%00010001
         sta $dd0e

Sets Timer A to run in continuous mode.  As soon as Timer A counts
down to zero, it will automatically be reloaded to the last writes to
$dd04/$dd05 and begin counting down again.

endless  jmp endless

For this example, we just put the computer in an endless loop.

nmi
         pha
         txa
         pha
         tya
         pha

         ;play 4-bit sample
         lda sample
         and #15
         sta $d418
```

```
We play the sample while all the code is still linear - before any
branches have occurred.  This is to minimize the distorting effects I
mentioned earlier.  The AND #15 is used so we don't inadvertently
enable the filter bits in $d418 with the high nybble packed into
sample.

         ;clear NMI source
         lda $dd0d

By reading $dd0d, we are acknowledging the source of the interrupt,
and the CIA will now generate another interrupt next time Timer A
counts down to zero.

         ;just something to look at
         inc $d020

         ;every other NMI do 1) or 2):
         lda flag
         bne lower

Now we deal with "unpacking" the samples.

         ;1) shift upper nybble down
upper    lda sample
         lsr a
         lsr a
         lsr a
         lsr a
         sta sample
         jmp exit

When flag is set to zero, we shift the high nybble of sample down to
the low nybble so it's ready to be played next NMI.

         ;2) get a new packed sample
         ;   then point to next
lower    ldy #0
         lda (ptr),y
         sta sample
         inc ptr
         bne checkend
         inc ptr+1

When flag is set to one, we load a new packed sample into sample, and
point ptr at the next packed sample.

         ;if end of sample, point to
         ;beginning again
checkend lda ptr
         cmp #<end
         bne exit
```

```
         lda ptr+1
         cmp #>end
         bne exit

         lda #<start
         sta ptr
         lda #>start
         sta ptr+1

Simply check for the end of the digi, and if we've reached it, loop
back to the beginning of the digi.

         ;toggle flag and exit NMI
exit     lda flag
         eor #1
         sta flag

         pla
         tay
         pla
         tax
         pla
         rti

         ;sample's lower nybble holds
         ;the 4-bit sample to played
         ;next NMI - the upper nybble
         ;holds the next nybble to be
         ;played on "odd" NMIs, and is
         ;undefined on "even" NMIs.
sample   .byte 0

         ;flag simply toggles between 0
         ;and 1 - used to decide whether
         ;to play upper or lower nybble
flag     .byte 0


Improving D418 Digis
--------------------

D418 digis tend to generate a lot of noise, because, of course, the 4-bit
sample resolution.  Over the years people have come up with numerous tricks
to
improve the sound of a d418 digi; here are some that we know of and have
tried.

The first, and most obvious, thing to do is to use the low-pass filter,
since
a lot of the noise is at higher frequencies.  Unfortunately this won't work,
since the filters occur in SID before the volume amplifier -- all the
```

```
filters
can do is change the DC offset that makes the digi possible.  This trick
will work for methods that use SID voices, however (such as Pulse Width
Modulation, discussed in the next section).

Another trick is to "dither" the sound, as discussed in C=Hacking #11.  The
idea here is to generate an intermediate "average" value by toggling between
two values.  For example, if d418 is set to '8' half of the time, and '9'
the
other half, its 'average' value will be 8.5.  So this is somewhat like
adding
an extra bit of resolution.  In principle, you can extend this further: if
it
is '8' one-third of the time and '9' for the remaining two-thirds, the
average
value will be 8.66.  And so on.

Now, we aren't _really_ increasing the sample resolution here, but are
instead
increasing the sample playback rate -- we're playing two samples ('8' and
'9'
for example) where before we played just one.  Don't get too carried away
thinking about "average" voltage levels (after all, there is an average
voltage for the entire digi but that's not what you hear!) -- what's
important
is how well the sampled signal represents the original signal.  If the
original signal is rising from 8 to 9 during the sample interval, this type
of trick will work well.

Which leads us to another trick: interpolation.  This is really a
compression
trick, more than a 'resolution' trick.  Let's say that one sample value is
5,
and the next value is 9.  It might be reasonable to expect an 'intermediate'
value of 7, to play right after the 5.  Once again, the idea is to increase
the playback rate to better-represent the original signal.  This type of
trick
increases the playback rate without increasing the amount of data -- and as
always, your mileage may vary.  Many modern soundcards and CD-players use
interpolation.

Another curious trick is to add noise to the signal -- that is, the 4-bit
sample corresponds to the original signal plus noise.  Sometimes, by adding
noise to the signal playback the noise can actually cancel!  The 'dithering'
trick above can be viewed in this way.


Boosting 8580 Digis
-------------------

As most people know, there are 'old' SIDs (6581) and 'new' SIDs (8580), and
```

$d418 digis do not work right on 8580 SIDs, (such as in the 128D, most 128s, and the 64C) for the reasons discussed earlier -- the 8580 does not have a residual voltage leading into the amplitude modulator.

The software fix for this is pretty simple: have SID generate a signal, and hence a voltage, for the volume register to modify. You can actually use pretty much any waveform to do this, but a pulse is the simplest, since a pulse wave just toggles between two voltage levels. Moreover, page 463 of the PRG says, "The TEST bit, when set to a one, resets and locks Oscillator 1
at zero until the TEST bit is cleared. The Noise waveform output of Oscillator 1 is also reset and the Pulse waveform output is held at a DC level." So it's not really necessary to worry about the frequency or pulse width, by using the test bit.

BUT -- it is very important to set the sustain level to $f. The ASDR envelope
generators generate the voltage. A sustain level of 0 gives no improvement.

So, to 'boost' a digi on a later-model SID, you can just turn on a pulse with
the test bit set:

```
    LDA #$FF
    STA $D406
    LDA #$49
    STA $D404
```

Setting more voices gives the digi a substantial extra boost:

```
    LDA #$FF
    STA $D406
    STA $D406+7
    STA $D406+14
    LDA #$49
    STA $D404
    STA $D404+7
    STA $D404+14
```

The moral is: if you're writing a digi routine, and want it to work on all computers, be sure to boost the digi.

And for completeness, using more channels is a commonly used trick to enhance
digi resolution on the Plus/4. The TED digi resolution (the volume register) is 3 bits. Fortunately, all channel on/off bits + the volume level are in the
same register ($ff11). If one source is on, the output DC is about half of the
level when both are turned on. This trick can be extended further to results in a 'semi 4-bit' or 5-bit digi table (the dynamic range is enhanced, but

```
there are larger steps at the table end than at the start).  This trick
could
also be used in SID if the sound sources were accurately preset, but runs
into
problems due to the non-matching SID-versions and having the control bits in
multiple registers.


SID Type Auto-Detect
--------------------

The following routine will detect what type of SID is in use.  I've
tested it on a fair cross-section of my collection of computers - my
NTSC 128D, two 64Cs, two "breadbox" C-64s, and my PAL breadbox 64.  In
all cases the code performed 100% accurately - but still, there may be
cases where it fails.  I'd be interested to know if anyone finds any
faults in the routine, so I can improve it!

How does the routine work?  I was told that the old SID (6581) and the
new SID (8580) behave differently when set to play combined
waveforms.  I coded a fairly simple routine to use the REU to sample
$d41b (the upper 8 bits of Oscillator 3's waveform output) for a full
64k bank.  Then I experimented with various frequencies and
combinations of waveforms on Oscillator 3 until I found consistently
different results with the two different SIDs.

When I combined the triangle and sawtooth waveforms and then sampled
$d41b I found that most of the time the oscillator was just putting
out zeros, with occasional bursts of numbers.  These "bursts" were
consistently near $ff on the 8580, while the 6581 was always well
below $80 - often $3f was the highest it would get.

So, the detection code ended up being quite simple - I'll explain each
block of code:


         *= $4000

start    sei
         lda #11
         sta $d011

Disable bad-lines (by blanking the screen).  This prevents badlines
from interfering with the detection process.

         ;sid setup here!
         lda #$20
         sta $d40e
         sta $d40f

Set Oscillator 3's Frequency Control to $2020.  I just randomly chose
```

```
this value when experimenting, and it worked, so I kept it.  The trick
here is to set a value fast enough that the oscillator will make a
number of cycles (so we can get a good sample of the values coming
out) but not so fast that it might miss any of the "bursts" I was
mentioning earlier.

         lda #%00110001
         sta $d412

Combine the triangle and sawtooth waveforms and start the ADSR cycle.

         ldx #0
         stx high

loop     lda $d41b
         cmp high
         bcc ahead
         sta high
ahead    dex
         bne loop

This loop takes 256 samples of Oscillator 3's output, saving the
highest value in location high.

         lda #%00110000
         sta $d412

Stop Oscillator 3.

         cli
         lda #27
         sta $d011

Turn the screen back on.

         lda high
         rts

high     .byte 0

Return from the routine with the highest value sampled from Oscillator
3 in the accumulator.  This allows you to branch based on the high
bit:

       bmi SID8580
       bpl SID6581

Voila!


======================
```

```
Pulse Width Modulation
======================

    The primary limitation of using the volume register is, of course,
that it is only 4-bits.  Pulse width modulation (PWM) allows us to get
around that limitation.

    In general, there are lots of ways of transmitting information.
If you've ever used a radio you've encountered both amplitude modulation,
where the signal is encoded as the amplitude of some carrier wave, and
frequency modulation, where the signal is encoded by changing the frequency
of the carrier wave.  In both cases, the idea is to strip out the encoded
information and throw away the carrier.
    Yet another possibility is pulse width modulation: use a pulse
wave at some carrier frequency, and modulate the pulse width.  Pulse width
modulation has several nice properties for transmitting signals; we can
take advantage of it to play digis.

    Pulse waves, of course, take on only two possible values: zero and
one (low and high, etc.).  Over a single period, a pulse wave will in
general
be low for some amount of time and then high for some amount of time.
The _duty cycle_ of a pulse wave is the amount of time it spends in the high
state compared to the total period.  For example, a square wave, which is
low exactly half the time and high the other half, has a duty cycle of 50%:


          _______    _______
         |       |    |    |
         |    |    |    |
     _____|    |____|    |____ ...

Remember that, regarding SID, a signal like the above is simply a voltage
level.  What is the _average_ voltage over a single period?  Since a square
wave is zero half the time and one the other half the average value is
just 1/2.  If instead the pulse had a duty cycle of 75%, it would be low
for 1/4 the cycle and high for 3/4, giving an average value of 3/4.

So the _average_ value of a single pulse is simply the duty cycle.  So if
we change the duty cycle for each pulse we can essentially generate a
series of average voltage values -- and since a digi is nothing more than
a series of average signal values, we can use PWM to play a digi.

To make this more precise, let's say we had a digi sampled at 1KHz -- one
thousand samples per second.  Since each sample value will be approximated
by a pulse, we need one thousand pulses per second.  The duty cycle of
the first pulse will be the first sample value, the duty cycle of the
second pulse will be the second sample value, and so on.  Note that
the sample rate is the carrier frequency -- the frequency of the modulated
pulse train, 1KHz in this case.

(Actually, to be more accurate, we need _at least_ 1000 pulses per second --
```

for example, we could use 2000 pulses per second, and represent each sample value using two pulses. So the more correct statement is that the pulse carrier frequency is the maximum sample playback frequency.).

The advantage for playing C64 digis is that we have much more resolution for the pulse width, and probably not in the way you think! Because you are probably thinking that SID has this nice 12-bit pulse width that we can use here. The problem is that the absolute highest frequency SID can produce, using the frequency registers, is about 4KHz, which would be the maximum playback rate.

There's still another catch -- the carrier wave is still there! Imagine trying to encode a signal that was constant, say 1/2 everywhere. To generate a "digi" value of 1/2, you'd use a square wave, half down and half up. So while the _average_ value of each pulse would be 1/2, the actual signal would be a square wave at the carrier frequency (look at the little picture above if you don't see it -- its average value is 1/2).

Trying to modulate a 4KHz carrier wave results in a piercing 4KHz tone, and a _maximum_ sample rate of 4KHz (and this assumes that you can sync your code up exactly with SID). So that's pretty worthless for digis.

            - BUT -

What if we could change the voltage level manually? Let's say some hypothetical machine language program toggled the voltage level on each machine cycle -- the result would be a square wave of frequency 0.5 _mega_ hertz. Okay, let's say it changed the voltage level every 10 machine cycles -- the result would be a carrier frequency of around 50 KHz. The point here is that a machine language program can generate its own pulse waveform, and do so at much higher frequencies than SID can produce.

Toggling the voltage levels turns out to be very simple. As was described earlier, the way to "boost" digis on later SIDs is to use a pulse waveform at frequency zero. Depending on the value of the pulse width register, SID will set the output voltage to either high or low. So all a program has to do is set up a pulse waveform at zero frequency and use the pulse width registers to toggle the voltage -- set $d403 to either $00 or $ff to toggle low/high. (You could also use $d418 to toggle low/hi, but this method should produce more uniform results, and unlike $d418 can be filtered).

So now we're cooking -- we've got a program that can generate a pulse train. The next step is to change the width of each pulse to represent the sample values in our digi. Remember that the duty cycle -- the percentage of time the pulse spends high -- is the average value for that pulse. But also remember that each digi sample represets an average value over the sample period. If the pulse period is equal to the sample period, then _the duty cycle is exactly the sample value_!

```
Example: let's say that we have an 8-bit sampled digi, so that values go
from 0-255, and our program generates pulses with a period of 256 "ticks".
Now pick a sample value, say 56.  All the program has to do is hold the
pulse high for 56 "ticks", and low for the remaining 255-56 = 199 "ticks",
and it will have the correct average value: 56/256.  So a program to play
8-bit samples might look like

1 - Load .X with next sample value
2 - Load .Y with 256-.X
3 - Set pulse high
4 - Loop for .X iterations (each loop iteration is one "tick")
5 - Set pulse low
6 - Loop for .Y iterations
7 - Loop back to step 1

Let's say that each "tick" takes m cycles, and the sample size is 2^n, so
that there are 2^n ticks per sample.  A stock machine runs at around
10^6 cycles/second, so...

    (10^6 cycles/second) / (2^n ticks/sample * m cycles/tick)
    = 10^6 cycles/second / (m * 2^n cycles/sample)
    = 10^6 / (m * 2^n) samples/second

So, for example, let's say we had n=6-bit samples -- 2^6 = 64 -- and could
generate pulses with a resolution of one machine cycle -- m=1.  Then
we could play that 6-bit sample at 10^6/64 = 15.6KHz.  That is _really
very good_!  In principle -- possibly using the CIA timers, possibly using
fixed delay loops, possibly using a massively unrolled loop -- this can
be done on a stock machine.  (I did try using the CIA timers, but the
number of cycles to set up the timers was too big, and made it sound poor;
I've included the code below though.)

At this point it becomes a numbers game.  As we increase the sample size
(increase m or n above), we _decrease_ the sampling rate -- if, in the
above example, we instead use 8-bit samples, the sampling frequency drops
by a factor of four to around 4 KHz.  So there's a tradeoff between
resolution and sampling frequency.

AND... we still have this issue of the carrier frequency.  You should be
able to convince yourself that the sampling frequency above is exactly
the carrier frequency.  So with the 8-bit resolution example there
would be an awful 4KHz tone running through the playback.  There are
only two ways to beat the carrier frequency: push it high enough that
you no longer hear it, or else push it high enough that you can use the
filters to dampen it down.

How high is high enough?  You can judge for yourself, but 15 KHz is
pretty tough to hear, unless you have good ears and the volume is really
loud -- so 6-bit samples are within reach on a stock machine.

But add a SuperCPU into the picture, and the numbers get _really_ nice.
```

```
Everyone knows that a SCPU can interact with the C64 at 1MHz, and
hence generate pulses with 1MHz resolution, using code like

    lda #$ff
    sta $d403    ;Set level high
:loop   lda $d011    ;wait for C64 cycle
    dex
    bne :loop

where .X contains the sample value.  But what happens if we try to move
beyond that 1MHz?  What if we put some NOPs into the above delay loop,
in place of the lda $d011?  Well, in principle it means that the duty
cycles won't always be right, which corresponds to some sampling error.
In practice, however, it works _really well_!  Consider what happens when
the above code is changed to:

:loop
    nop
    nop
    dex
    bne :loop

The earlier formula still applies, but now using 20MHz cycles:

    20 * 10^6 / (m * 2^n) samples/second

In this example each loop iteration -- each "tick" -- is nine 20MHz cycles,
giving a playback rate of approximately 17Khz for 7-bit samples.  Which
is TOTALLY COOL!

And it can even be pushed to 8-bit samples (although I personally don't
think
they sound any better, at least with the code I've tried; maybe the code can
be improved).  Using loops like

:loop
     dex
     beq :done
     dex
     beq :done
     ...
     dex
     bne :loop
:done

it is possible to "fine-tune" the loop tick to somewhere between 4-5 cycles,
giving a playback rate between 15KHz and 19KHz, for an 8-bit sample.  Pretty
cool.  The code is also a little more involved (with 7-bit samples we can
use BMI for the loop branches; not so with 8-bits).  But it really is
possible to play 8-bit samples at 19KHz on a C64 (plus SuperCPU).
```

```
Using two voices
----------------

You may be thinking, Hey, we've got three pulse waves to work with, can
we improve the performance by using multiple pulses?

Let's say we have two pulses, P1 and P2, with the same period.  When both
are activated, the pulses simply add together -- that is, the total voltage
is just the sum of the individual voltages, and therefore the _average_
voltage is the sum of the individual pulse averages:

    avg voltage = D1 + D2

where D1 and D2 are the duty cycles of pulses P1 and P2.  In the simplest
case, this gives us an extra bit of resolution -- if D1 and D2 are both
7-bit values, say, then D1+D2 is an 8-bit value.

-BUT-

Consider, for a moment, what would happen if we were to change the amplitude
of the second pulse -- that is, let's say the maximum voltage it took on
was 1/16 of the maximum voltage of the first pulse.  The average voltage
would then be

    avg = D1 + D2/16

This then gives us _four_ extra bits of resolution, with each bit to the
_right_ of the decimal place.  For example, if D1 and D2 are 4-bit numbers,
with D1=xxxx and D2=yyyy, then the avg will be a number like xxxx.yyyy
(four bits to the left of the decimal place and four to the right).

Of course, we can change the pulse amplitude by changing the sustain
setting, so in principle this gives a very easy and efficient way of
playing high-resolution digis.  In practice, I have not been able to
make it work very well.  I used a sustain setting of 1 and split an
8-bit sample into two 4-bit pulses; I believe the result sounds better
than 4-bits, but certainly doesn't sound anywhere near 8-bits.  My
suspicion is that it is because the second pulse voltage is not really
1/16 of the first pulse, which corresponds once again to adding noise
to the sample value.

To find out, we can just measure the output at different sustain levels.
The following table gives the voltage output for voice 1 using a pulse
waveform at zero frequency and volume 15:

    Pulse Width      Diff
SU  000    fff    000    fff

0f  6.34   5.29   .08    .07
0e  6.26   5.36   .02    .01
0d  6.24   5.37   .06    .05
```

```
0c  6.18   5.42   .03   .02
0b  6.15   5.44   .05   .03
0a  6.10   5.47   .03   .02
09  6.07   5.49   .04   .02
08  6.03   5.51   .03   .02
07  6.00   5.53   .05   .03
06  5.95   5.56   .03   .02
05  5.92   5.58   .05   .02
04  5.87   5.60   .04   .03
03  5.83   5.63   .04   .02
02  5.79   5.65   .06   .02
01  5.75   5.67   .05   .02
00  5.70   5.69
```

Voice 2 is identical within a few hundredths of a volt. If this test is repeated using voices 1 and 2 simultaneously, the result is:

```
    Pulse Width
SU  000    fff
0f  7.30   5.25
0e  7.12   5.36
0d  7.09   5.37 (!)
0c  6.95   5.46
0b  6.88   5.49
0a  6.78   5.54
09  6.72   5.58
08  6.62   5.62
07  6.58   5.65
06  6.47   5.70
05  6.40   5.73
04  6.31   5.78
03  6.22   5.82
02  6.13   5.87
01  6.07   5.90
00  5.97   5.95
```

Note the weird step at $0d -- the response is definitely not linear!

Now, to summarize, when using one voice, the "positive" amplitude (about the mean 5.70V) is .64V and the "negative" amplitude is .41V, giving a spread of 1.05V. With two voices together, the amplitudes are 1.33V, 0.72V, and 2.05V respectively. If the two signals were simply added together, the numbers should be 1.28V, 0.82V, and 2.1V.

What we originally wanted was a signal like

```
    D1 + D2/16
```

that is, another pulse that is 1/16 the value of the 'full' pulse. 1/16 of the positive amplitude is .64V/16 = .04V, and 1/16 of the negative amplitude is .41V/16 = .026V. A setting of sustain level 1, on the other hand, gives

```
voltage offsets of 0.05 and 0.02, giving approximately

    .64V / .05V = D1 / 12.8
    .41V / .02V = D1 / 20.5

So, in summary, whereas I wanted D1 + D2/16, I was actually getting
something
that varied from D2/12.8 to D2/20.5, even if the two voices summed together
correctly.

There may still be a way to make all this work right, which would be great,
but I'm tired :).  The code from my attempts is below.

I also could not get two 7-bit pulses to sound like an 8-bit pulse.  I took
an 8-bit pulse and divided it in half, assiging each half to a pulse
(and giving the extra bit to pulse 2, if an extra bit was present).
I suspect that another issue is that it is impossible to update both
pulses simultaneously, meaning some delay between pulses, which translates
to adding -- surprise! -- noise to the signal.  Perhaps it would be
more effective at lower resolutions, however.

If someone has some success using these techniques I'd be interested in
hearing it.

SID lockups
-----------

Blindly applying these PWM algorithms has a way of locking up SID -- like,
locking him up hard.  To be honest, I don't have a good explanation for why
this happens, and I haven't yet found a good method of prevention --
toggling
the test bit, playing a real sound for a short time, toggling the gate bit,
and so on, just don't seem to "initialize" SID reliably enough.  Sometimes
the
code works, and sometimes it doesn't -- it's the same code both times.
Often
resetting the machine will make things work; I'm not sure what hardware
resets
take place within SID, but the kernal certainly zeros him out so that's a
possibility.  The other observation is that playing a tune seems to 'clear
out' whatever is blocking SID.  So there _must_ be some kind of software
solution to the problem.

In the example code pressing RESTORE restarts the code, which will usually
clear the 'blockage' after a tap or two, if it happens.

If anyone has some thoughts on this issue (or even better, an explanation
of what is going on!) I'd love to hear them.

========
The Code
```

```
========

And that about sums stuff up, we think.  All that remains is a zipfile
containing code examples of the things we've discussed.  The archive
contents are:

digi-$1200.o  pwm-2cia.d.s  pwm-jmj       pwm8.d.s
digi-$1200.s  pwm-cia.f.s   pwm.d.s
pwm-$1200.s   pwm-cia.g.s   pwm2.c.s

The main example program is "pwm-jmj", for the SuperCPU.  This file contains
code at $1000 and a sample at $1200 (the sample is a sample I made on my
Amiga years and years ago, of Jean Michel-Jarre; it's really not very clean,
but it still sounds pretty nifty).  So, to run it just load and SYS 4096.

You _need_ a SCPU to run it; it plays the sample at 7-bits and around 16KHz,
using pulse width modulation (pwm-$1200.s is the source code).  If you want
to compare it to a 4-bit $d418 digi, just load "digi-$1200.o",8,1 and
sys 4096.  The other files are different code experiments -- using two
pulses, using the CIAs, 8-bit samples, and so on (all written in Sirius,
of course) and may provide ideas for anyone wanting to experiment further.

If you want to try your own sample with the code, I suggest converting it
to RAW format.  RAW format is just the digi, with no headers, and uses
signed 8-bit numbers.  It's what I converted the sample to, and what the
code is designed to play!

Okay, enough talk -- go listen to that cool 7-bit 16KHz JMJ digi!

begin 644 digibin.zip
M4$L#! H    !  -IQQIT41>R90   &L    ,    9&EG:2TD,3(P,"YO ""D
M E!(7RH)A?8-+,0OU;]&%:@Y;&1!(H!&%R0^:(3!8B, U!H%"%E XD,#$H,T
M(D -3ZL(T%)LA'E*7Q) 2G(JX4A-E(=6'95!TV=.'[1TYO:5VI<,#+1P+Z$E
M@- ()A8F 0%02P,$"@    8 _&C'&I6%<YMA 0  <@(   P   !D:6=I+20Q
M,C P+G,,"@$3 @,$4TH%:OO[6P8  1;W]_>W 0)4V5 E2.))CR8%610KU*)2
MDS8MZI2J3I#0KS=O@,W3F2L'&1/F2YDR7\*$ 7U#T$^E'@5),B:,YAM"?9K4
M*570/4&3-!JTX=2@3:$R+<+,,65L<U*G2:DV!,V4:%"0(WELNR'HJ52#@E@G
M:_NP8B%6[\6*AS:X>I)&C8+?:3PI4ZI%I4[A>J(T8];H*JJV0840_JCV!M6D
M)8Y.*"V4/(92I8)AI%J:,<'OA%Z5Z=2B((4^?3J5*LBG3LB:!NFG%DVB;I)$
M8<:H?%"G1!0^H>:&3)\^A5+N+1I7[-H0=-&G4L['(?'/=*H48@2_LWG0I45!
MT@0I-"G5*5;=@.*QP6.Q/V;1ACJO!DU*!==E(]&B6!N"%NJT*(ALJ#^ITZ$@
M6H.U!GY1[<H8$+^8=FA3*)FW":,M,!Y]*L7ASC%A8#P.>BB3GG<E^57L 5!+
M P0*  0 !@#Y:,<::WXHL L"  !X P  "P   '!W;2TD,3(P,"YS3@<(K0/]
M'0,=" 5-!@@M!P,$!0@&" <6" 8%*!T$!00% Q4& P<&! 4D"!0%%@@&!1@-
M&/V-_OY^!/[^"!X(+@@.)P8>!@X'!@<>!GX$_NX/ 0(# @,"! ,T!AL%N_S\
M/ L  P(#!P47!O?W]V>1(Z?_]!?HSN]WXOK3[=P9L\DO)BKVPS8Z;/BWWDB5
M2J1[04MZZ*3=OOC3M7L87T)[VK=T0\)8?&MJN$"">OVT9$K+G&C];X8N_ZKK
MM?_WPQZS37!-M;#\=LGUT\'LO6M%MWO<V=],Q>6%4KTI=;D[[<F^^&'/YZV_
M:;/+JQ;3;MI'EN"!%59N2&N*JP3K"@SAYY8]HMNMT16'-Q=66&*5P%GA3%=S
MO<>4DBA-0I2("P54CHCJM+4NQR1,[7G#*!.L%YNS2K;\;-I4BW=%ZHN47QFC
M]J>WF7%^3W!I'@?4VB=(/:]!_.8,G?XIMRZ'"2SJ0=AO=WX\1>7;KB['+HGG
```

```
MI;[2SK:_R2YQWDXM<#93E^>!Q%DBM[\IG6!><INX^\\5KLO1;([[H%_2SB?A
MA"MUF>VNL3[N*&ORM#;CT'_$%$^&HX_GL?Y)&BH/5M"OH".>+_43^K07A_$X
M_%9P3A7U: B<9J4AKS(-)IUS/=Q'O-X(/ K^RJ7S:.B4TO72>=S,DX_C" P=
MRJ;?SR;JF57<EG+[\5@2(3>A6-7#7%!+ P0*  0 !@#\:,<:]D)"*[$"  #P
M!@  #    '!W;2TR8VEA+F0N<TD'"*X#_AX#'@@&3@<&& X'% 85%@<&!P8%
M!Q@>! 44 P0&!0,'" 0%) @4 P87!@4^"/YN__^?!?__.#\8#P@''P8?%Q\'
M?P7_[PXB(Q(#)0P%!PP$!NS]W04+$003! 8#1F<&]_>G4:-&^5%>H"?NO!O3
MW/KD;K?C7W3(,!_V0@<4\%LB)T5>%\X-E3_5"W?KZK3=BZUN]7;^OL "":)Z
M0UL#?V)PA;-?T\_9PCP%_92_VBWDY^8I?[I;*?K/M'7<2UW6=?V_:4^W,\%F
M4H?Y5W.NZ2_F)=AN/^\IS_F)I93RFB[2TI0W7EW__W1G_TP_SB\$[,]4>WZI
M8VE--"VI:T*Z6-_J[A>N/=-]BW&7/G(NN?IP^GUKW5D$92#O)=)W9=4\K9>R
M!XL$7)<Y70P?6^UWNC?-U264A+)[CZI33E(M5O&?FT\'6R*X3'&+T\Y:@V$-
M%UA+D*SAZ1L-4R:549+;,W $+BWI8G1L7UV37B)0WK*8]Y+P+887U!RAZ=NP
M/,G#<#G.K.&!.\W9OFT;VY[)FS948/*]&H5A@L<1Z;%Q:^)"RE-*%S>$]?H%
M_J8VMMM,]ML]]Z\!6)6LM:/<L[KW%6H6U!9J((KT[3RF+!8%RFXR^F8A+OT8
M(VT1?V:\$W#6^Y)_YG2B@ZF6><HU;/Y0O=#=%S]9>WC2']ON?LWE2Y%*NNDB
MY=?[=;GS_[[51EZ15:))7B$I[_U/X/MT /'"#>JYI'[Z&2DSEKW,U)]T,64A
M,*S:,0ANR^DB3&C^Q.79] ?XQ1=E&GP6HE;O"2LMY?>C&I;SCB*'&&2#L[6O
M:HSQ>)O;C16>N?K.X^GB'H(WAH-=\6'GLS6<(GF$NO^CN84Q,(V%@$>5/%[P
M'(;0#,\9@Z*@U,7L!RUNID1R,TU:(A;@M"C\][I?%I,54$L#! H !  & /=H
MQQKFO<<O:P(  &D%   +    <'=M+6-I82YF+G-'![T#_1T#'0@&318-" T'
M%!8%%@<&%P4H'00%! 4#!!4#!P@$!00#! @%% 4F!3T(_3W^_LX%_OY(+A@.
M" <>!AX7'@=^!/[N$0$# @,"$P(')28,!@3\#/V]!@L  P<3!0,G%O?W]S>1
M(Z?\E!?$*9YWH[N;._G;-^%?5*BP'U;#!!+H6R(E1%H7SH8NG/;%WEQ[=O^R
M[6U?4[@OR,"";M^X<\#'AU0^S>N&.5.>I^">\FVS(<_/4^'TMU7X/[USQ9>Z
MK.OZ?[=/WZ1#2*>.\V_G?#=<3%WC5MT.<S,50(^O)E?KBLTE^94NS___]*?Y
MZ:$TOY#!KRNYV][FQ>XSW;?\=-O,+[547:L?V:H\K9?2HV_ OFG69:XKX;)M
M\T[_IKF]!*>X1'>75"DQ035*W30A:MC+FG'4V:X2O"NT_']^/KTZ$U1X&9U8
MX3)K@F&%IP\Q<)42-2E@@IR=QQ'5\:A<7>I*5'9H+U]V\"B#9W,O<1^O((2Z
M&W:,U7PYQJSP"!2O.MX)E([H&PUK&:P098JO*PV>>^X%]::=[1L:]]W,P^LD
M5AE'MUAT&L[7\GC8<REY ?-8MDGHS2#9,Y2B)5\B_-_O-M-/'!XGU16Z2^$4
M"Q3."E^Y:$'<[,,X+]4W9:9!QSRK1JNX4E>$SS4/96UA37^_.T2@%H^-,PZ5
M'SYF/DT:+>J4M(6='4JO3+'+L=/S;5TI[EUZM]]9KB[5(IE$+)+RGF!./STU
M?7^:4MW"Y?X^C+ K)4%<+'G*=SR.!JZ-<(N;>]+*=H-<;[ /RYAP[BL9^T'#
M9DA$-H-)$Z',8$;A'W1/I-(44$L#! H !  & /=HQQJHW6A*AP(  .0%   +
M    <'=M+6-I82YG+G-&![T#_1T#'0@&30<&#0@-!Q06!387!"@=! 4$!0,$
M%0,'" 0%! ,%" 44!28%/0C]/?[^S@7^_D@N& X(!QX&'A<>!WX$_NX0 0,"
M P(3 C4'%@P5_ S]O08,  ,' P8"%C<&]_?W)Y$C1_J17A"G=-Z-SM_<*=RN
M"?^B0H7]L!HFD$#?$C$QXKIP-G3QM"_VYMJSNY=M;_N:XGU!!A9T^X:= SXA
MI,II7KZ?,I5I#.XIWS8;\L(T%D]W6X7^TSM7>JG+NJ[_Y_?IFG0(Z=1A^NU4
MR/<74]>X5;?]U(Q%T!.J*=5ZMJFDO/+EZ?^?[C0_W9>G%[+UZ;:97FJYNB:8
M!J'7<^ZVMWFQ^XSW+<:N?F2NKO4\G+ZY.5U:!$H#N9?X[U*K\KA>2@\6=M"Z
M3/4<+MLV[W1OG-I+4(I+<G=9E9(258/UI@D1(&7-,"#YNTKPKH"5_PO3Z=19
M8N%E=&)%*JP)AA49/\HP)<XPG@CET^ W7%WJ.2K;MU?!-MT8;Q9S+W$?SX<1
M?.?['6,57XXP*S( >5YU=B*EXWV#8JV&%:Q,"?7<&#CG7E!OW-FN$3*_FZE_
M>8D4R-$MYNG8GZ\UU.^IG+* #<DV&;T9)'OZ<C2$KQ'^[W:;Z48_YR<8EXG4
M G,RILX>=BF>4O'+5,%:"!>UR&;.2_6-F;&',\VJT5(6J6?A<\V#2EM87TJN
MKDG>=SN/"'0CL7$&(0D?,YTF#>6JC9NVN+-]^56H<B5V?+ZMY]+>Y7>[G=5X
MB0MC%"57!#R?;GQ@VOXTI5JY7.GN W#CZF2(BB6/A3R)HX%KH]QB93YY9:M!
MJC?8!V1(./=E#/V@H=(GGDIOTD2HT#D8_+WN24JJ E!+ P0*  ( !@ *:<<:
MTV5I'GL[   "4   !P   '!W;2UJ;6H<  $" P0%!QD'"0LN"QX?"P\+7PD+
```

```
M"0LO"P\*_X\< @,")*4$!01%!@<&!R8'%@=F%P9'!@<F!08%)A<!0DR-L#%!
MZJGQ-L8X,#7$QAD('V!A_:DE%MYWN/D-?F. U!L'$/P"H/^_L0'"#Z(Z/U!]
MZGA;1X2> O$(_[3^I("5JD[Z%S5\!:);('7UZA%#VUV)8^40]YO7#[V[^?VE
M]R<'"[VZ>Y#0:G=""+4;*].'$%L'Y YS,)?_'PB2_']XH.\4NF/.HB."]!C1
MM42(OR[M3X(_$#W[BA ARV_L%WIJ+HA^6O4..&W4%D#X QL/G-[X+W3A(%,W
M!/@(]0D^%C"X"0QN X,;P>!6,+@9#&X'@V=[MV\X9^H A9^H\!\4/N.<XF&F
M)M@8<<[&BP3?<4[L./>+_J\9Y.C^("L(_M9_F;/ U#8+!\0*<GU$IKMS8MPO
M'*)XD**&;])#F+1:/7KP *%,>C"H-M>>$0<N795[L08/B+4!Q!IN31$BTM5,
M(T(@7>L/$O+-R5.S3@5)DNA,&(Q/]2E(J45;%G5*M:A4@*EOF#]G@ B&/4CI
M_T'^'@]R*CG$J_Y)U8-Q&*?BYTJ7-,# K3 ;?DKA,%L_MM@86?2#;!WI\)5'
M;-#X#2%;A!]ATQB@@$VK?TNV"&W3S1KPO81]9\#-8CU7@9#I[V>2^*W+WT\>
M&3K#U@I^F^; U@L*O^9 L50K7OSHT"-#BP\K-ESIL^G3D /9OV>_GKSX\.'3
M?R]?'OWY]^C?V[^_?_]^@/L#_L]'^ 4"OZ7,B_/?I]<NFC;V]?;9WY=?O___
M@@L3/HR8<6)'C1Y-;DS9,>3'D" [G@1:=FO*A?7GMZ\__ORWP!]7^.G'QR]/
M/W[]_/?S]^^?;_#W#^P/L.#%G1[[B;YWU;ZSV^]O/^!]@/TP?<*&#"<^E!C1
MXL2-%"]:K#@18T2,%F=V]2I3X;[<J__WY>>?5W\>_;C G3[]^_;RU=>7_QY_
M_7^:'F#"B"QI+@PHGSSPX>-6?;])7S\@__T&^R<,N- @PH<)+S:D6)'BQHH;
M+X(LF?4L3XK]]?>C_?KMUZ]/5_SESP6>IF_?OKU[]^^6K_+OW\?_W__&DQ7U
MO_^^NSCX__W]__]_<"!"A!$=-ISHT"'%!:/M4.'"A@6N7\'@[WR:<^#]]O,J
M/?[V[^<);G:)/[]>N)\O?WT\X8YW^//GR\\??R!&C?OG=Q^F?/;NTF5_0 '@
M'N'"A0\;/GP0^ R:M\2&$2<NC+BR:M"-_.WK&]5N?]VE)[]^O/MTQ-?N[45?
M<./KS_TT2[\%/MRX$/]WY,6SF3\]X,Z':1]@[8@50+L":KX [B=$2IG^X,:>
M/SONEZ?9[=E9^_/CRX_#].X0'7V^I^R,#[7[V1>U[]]_8<:+_M-+;]YMU^_/
M![[S^<(#'FQX\&%!A@\V_4**#R\P?X5V[_'G4YX@%28D> "\\UW[=Z?U*),C
M/$T1XJMC]_5T[SX;-,BP [U^L+WY^0$1*DCM@@4##A!W@X=XD+S _P0+(DSH
M\(!J5E@19<^<(2<2R/S^ 0Y;'JU4*\^_QRB.)U^>SMQIZ_@^W7Y9^_X,%>('
M7UW\N]0*W@83+#K.0O05*SCZ"+$U+A<4\/[\^Q^\>%*E1P6,[/WX M=A[]>'
M[??5ZO?G/2]?/=_R#>KSW;7Z_ D9&EP?GARXB_]^MW]@>3'?M?R 6,-/XA+S
M-5^C3R#+B^SX4$#OG3?[=(:B0U&AV%[QHOT RC[A ?F)M?E]\P779\?NK=Z)
MOKK/<VW_!@^@EJX^\2+&C1LY=KQXP94SU.+YE"<^]8#X#"5.7IP[;ZZ4$]8"
M#U%<WUZSDVR!!:+=> <,D"P_]"]O\J"A]0VNY*O\ 8=> .8RW 3H[8(%"O6,
M(Y:_Y,8(MQT@TZEV/$@977XZ95['?\V/'[^^><_/[=L M#\ >LJ0D"=M^5_@
MM?+^ ?[KCS2E64[<8<L:02R_R0^W/ %A';]]^?':(\'3E,!!KZS62LWR.&5Z
M^+(_V ^H;[CY$1_B>?,0G+T$^%@AP@.D)=_[-I5B/8'@EDA1(@/4]_<KXI?_
ML3W[TUR)+$&5Z.H$?A=XSW^/\>OULSQ%]J[Y%3(8YRSY!H%C!XDK/TD>;L6R
M*;#R_09%CM:C.ZU')[OS\TM-A4E7,MP%8M/Z+:_,;2]SQ)G<I625L+*#1_$!
MQ@MW^4S$\(, ];'"Y3';\P+'] A]OU#SOU4390=8-1*UZ6GK^I*O/45.BZ#Q
MA9I..\$T4-R?8\B1(G!>1B_F'_A/4LB^S9-35IOS^&*-[(M2>N87"%N72?\8
M388'6M7:?]9'F)X5>7+^J=)+O)@Q@[@MT>%"!,'R?W[!$:55G.O7HP^A6FF?
M\UV.[;6+MIVU9[#9"P)?H8+ 3V"])F7__I9GA0W639=K<M6Z9O<*5#SJW?<N
M5='6?\H&^UM4JK>2]P%H"]1@@P:(XX ![+XLUZY%FRX7\C1KX[M6NG@2S?Y,
MX___,.-^^[:[C3N[83B]F<?]HDV=9@>_4,"JPH\0>/H<VS>"[/WOT2+65FN4
MXX5ULI0E:E.C);LZ8X,$ 80"<MP#YD>H(>G-?_\^TA&[^.%$"CG/$2H!<=H#
M_1WK\MU!"DIE1&CCWU=S21_87^WMJF95P.9_L8:.58I,R<?R9Y!IAE"\5#L!
MJ(H7*IYKC^#C^?/@HW3C?[DK\^\1 PM(C0\,*QD*&P"U,K>^\5E KR%AZ(4+
M4@XS3-"YX%<MX'S.8_KT5#;?M\<XNN7_Q4,V$+B:!?F/KH$-9 N)+T(E:;;!
MF%^O+B\>BE]1K%!WR%A@?:\F4IM_/XZ/5#U  800*4*Z/D]A!=E*C$^0(('+
M+4 [&K24-[["T?/S\[9"A?-YCVYS@*S/ 0*GWX1O18L%#+/4!50M-+CJ<P!Z
ME;*1R#\>)ZO0[$\0@)B^"1#.!M(9S=K\^?E&S%&]G0 ;P \&IT]Q ]14.ED:
M;0PU7<U#SH<;T9[.JZA? )/Y0/V2MQ*] JHC9<WC4L"E;T8"CQ6DFX29#0@+
M#1OVA\C4<+3]]NW;\U_'0/LJU51N!] V_E?,J!KX@*%B!-5 M"Q(T;>E.NMC
```

```
MAXZ'RF0W[@X8,.#,UY[N] ,VQ#CBF:YY<V?/H$./)EW:U.E3ITV7(B4*=.=-
MF2Y70)U+AMS@\).JZZ]W.S7-O5]_GKQX\-Z[<]^N';NUZM.C0WONS'DSY\Z=
M-VO&;/DQY<F1,V\.33ESZ,Z8+YL.7;KTZ=NK:P<_GKOX\^7/NW?;A,?;=#0]
MT_XHLF3.G3R11FUJE:K3L%NOEG4[=N[=NW/Q_OT3.,#SZHV+5FU7L%RO1G6Z
M5&?/FRE9GCAKA?T?/)QH>F;CZM:O-V^N7/CMW:E9ER9]>?/DQ(T!!QJW 04.
MP <GRO6F6[NVG;OW<.+'ES>?OIT[N*"UN+@M@]\W">0U=<Z7-F4T&X<UC"]R
M\%A['*%R1VG29<N6+U7.3'&:7R"^RY0M2ZHL>?(DR9,C;#OE"KI=Z'?TP'3W
MF"%D1C*M40/$O<+@6CE/(!49?8GMDE?.<8;/!5,3U%>^C/G29@Y"72C2I4^K
M5LUZ=6O7L&'!ABU[=FU:M&C+FA4KIO'+EE73^&L"/^U:M&G+3'[:L66_ANWZ
MU0MD@TG\L6._=KV:->K4I4F+%@T:U&=-FBP876<I'4_PEF!E?/M-;JD<'OSV
MZM&-)_?]NW9MV*U7DSZEJV>V7-GQXT6!CSCQX$*MZ35 "B9\&''DQYDO96NI
MO5I6X+M^'?N5H9Q+MH$'#Q;XRI$[7^Y\^K/PKP[].O;LX+NQ2C5+1S=V\#ZN
MIE08UBRO.4#%):*%JB$N6,!19)@"/X+*?N/'CR-P\PKDCMDBL9[SRY4L41RW
MR90H5[I@W#-?THPY\^9,';KRCE->Z-&C3$[24:M:(2W!'GNF,K]]^W;NW+IU
M G=<.8KO]@WDLTV;ENP9RQINKUE@I5CYZ1-M?LG@;F*XD2(A;:% ?_J Q'OB
MD'Z9,XJU^D/G\I8X55,G9\GJN][>VGSX[MRS6Z<^77KTY\V?.V?>?'GR9)NR
M<-^];]^>/?NU[->M7[-^O1IUJ4!XY\^;^OE,@?#-GCV'%DVZ=.K5J5MM_NO7
ML6G3IM79MFW?SHU+\&_3FN]:@<.W"D>6CPNW>1W#?UWDVF7IU;.]2Y[Z^/#B
M;AJA4S)E\N0:?US@"*[*;RIK=[ ]KQXHZ1O?V'. K"N"'L$BJQ2PG"'P-0@N
M8\D1/K*D212QC1(%;J<8W"J$2S?'I&FC&^P3Z-"A18]L2[<5#AR5ZM0I@4?X
M5*UD5JTTN*]>87RI3YDF-<(K_]R)T\;KW\R9,\=ZX\2)(_%KSIQ)0_!,.T:E
MSE)EBHC/'3<8<\4LK6$$$@,-/-Y7T#A?X,8GD^R>O?KTYL=_WQ93KTY=^C/?
M9KX<N?'BPX4'!_[;MV]?K?+P88%?_'AR9?D^^O1IMV#LV+%?OQ;XUJM/2R_4
M/W[<6. 7+Q8X4./&C15.L+^<N3/8KWX]NW;MVK?!MC?7A-KERJK3IU?7,MW.
MJ'E=G-7B,>YU6:X03ZJII8Z/YO* @TX;"V%=35Q'R,*F",(I[A"X.005X7:(
MP,$C29JP#;M\@9MG6-QOXLPQ^#<BNV8-RGPC<"%>8WVR7P2>K>0/-#B=+LI5
MN9!%H>$1=?8;1M--\H1V0/&H1D)\R0$!,)J&FA^$588@7"E3,)OS 21J<!3W
MXXI_YZX-LG +,V?/%CAB]]T:7H8B=??YYP('C<KBQ7T*D_NW:O<X>!A["0<5
MW)L%[N?24'.VT[Q-6Q!J]+O/(0N_CS=200V9WP7>6M=M.63J6)DLWK]/GJ"7
MH9I& ],&"!87($GHJ9!1(0O7%2T(%\X+[MG$N\/-(@87W%\C!NS&(/6DR(-@
M/+RGP6S!H+I"X IQ]+K&P21+EBP!.U+\D2-D%VI;I&#,II :*ZCI"#4<,$!$
M)-7 #^*J4!M!*J&4B0'ZXRI4]7,7*C;MZ^IV8W(E4>+&[P(7TDAIB-QT*37.
M]"(ZXPN8M(H"%7S>?'EQQ"+:^?PY^QYW':65?&SQH]OYCB*2G-GLV;<[#14N
MV<C/O5\S(KDFK9(^1[$G,(O2S?[N.NDFM$%J#7$B60S1/6V84R!R. .EJ+%F
ML(1Q?G!M_/#!EUG%+JF>$(%-0W2#V#I5=7E8U!9KV6)OB15:(BE+"1BF0+BR
MLB/X.(+(HK&U^@+Z'&$DZCUMUMJP +V*+(<!?!2!NK:N@.Y_M89<@;S>\N.-
M$\+)%)"8OCT_#*[GC()W0QD+ (AR:/1N$997462[^Q365VWNOR^H %/W]_(*
M=42$U:'I%*C$TLG#:.]D+#9PAS&#6U6R6D&W,H%@4E2A1]>KQH\_GU*C^.LR
M!>@.XO]TB9"*'50T#WA)9N"2D)S+#1J#2(5^&LFB$!K?QP=4D2IK+ JJX ,6
M+BK;,C(E?Z:<=M(ILMOV;U]=(9V.6[^O4&.%\;Y8\D.EUU@21*,^4D4:1HMZ
M ]&WV#1JA@YTB@_@7<.L:<\_X#R)<SMPI^_E[XLR/3)I2?WCN8<OO[UY<>71
MG3&\GU._O@W"BO/LY76@S0Y%4[_U;XC@N;W"HCLQ(%KV9@="IL!Y#+,J8%[!
M8B^ 1FIU5K;KV?OVM'NC\U3]K3^>?%DS)\^>.W7BI"E#Z7NE"4V8PVJ;7M"[
M,=:<:I."O%T P?0_:/&]$+WUL()^)Q)Y??Z0#\>G B;&73SY]._H4OV!.'L1
MD[0.MZ.W<]&B0X,"];FS1@OZ"O:;0"^P0YLQ%[:K]N&];1Q;G]YL&:,X!__=
M:^)D^+MKN\I9EP:M^?+ES*0G0>_+DT??;LWZ>?;ET7S^V:4A*&\'1ZVK>__R
MWKBUK%$R-$$&EWYGB>Z>(*=^L@7 #9<*_"808[)R^Q C(/<(_IV2\=Z[9\FF
MW;*P?M(P.*E0HUO%+F6!I@9G5F5THY[N_:<I]9\S+XY\."ZQN6E+5?5$Q@C0
M@0L%/F34MUUWUKQY=.I4!*-H']M6-'[,9A<VUYYM5Y^GHS[_&S;*Z4JPRWER
M37)JZ !)TY Q<E!Z-FD2)$@0(V\1.?.7,MSZ+N[61:U.+4K#ZA1)HRS1]<AD
M7X(>HWF+]$"-R5F*>HKDBLX$ N7^%5/.W-$U:_93JU[5FD42>\3)9.);LX"5
```

```
MLWK=NO7KUS*#]%:M7=.4E>.:0<62Y+5RS9HUR^B;;=L&E6ZC2Z-]4YGCAM4Z
M1;%<M5=]J)''7FJ4B'.A4W2[N=D]O2)1:O\&""@?./MLKT!5N/S^OOKMT8LI
MTK40<=2G:K7 '_V9<Z;3O/CPX<.%!ST&U-#C)XK\PH4'$Q+\0M\(V$",GSB1
MZEX<F7+GU+:44L []&C2IF@I2\JC]C'N%98&7AP:Q#T\]6:?8M_%B:4,\MIG
M@UBRFYE<C4VK"[RH]#LU0W@8NM#%"9(RWLA"F*^!5TK-(HJFH$I#,XP\"K8C
M2LH:LYB3V"1M3&Z),^A2)V"ERQ9[UHQ2B;#9JE6;ELUB' <8IUM7CC$FN.O>
MY;,J[%>/XL^-$[CGOFWS*H65L5ZUFF4B_[7J4Z- >2!4N4*5IE/B'1>#:[XH
M1O%_A[OM#)]*[O !,99"MDC=A'I[<GVE9DR>CS82)UYLW/?6O:L2,\1;S;-9
MKUK&_JU:S!6_-8G<NS<O+Y=KB6.7'SOVK+$2Z+-&G?J4>:=&K3JU:E;;&.7+
M"JA-HP(?%S$6>''Q7($?["J/3+5V=^]?OQ6X?PLC_&#R%#-E3E+Y)V_&\J7;
MW>BED&7TZ;8VP5S-:>JRV&$$-@T5[@E3DR]>,-GN"((VTDW&<0QO#Q3(13P(
M-("-*ET:-6J47725P&LL:[EK%VZ9B^&#B8BG\8B?&8F/!8,O5XUR#7UT2.V)
ME3P+&)5)RL\E2;A-B523&,T<'MSLHC[</RASA<=3KY!U">OV9.@-:0!_6RJR
M"'2A_63X:N+=;&)7*:Y4L1.\8Q\-&H^^+!=?EHCZZLI%2B4EWB]JEM1=?ETI
M>6>#GPS?R6+EJ'P?71KL\;ZR:>P?6Q8#T\'(M8-AIJ9\\N3&GG*5/TP;!;U,
MZ_MRKY;$3G^C&/@</*%M(HI,5[!<+HU8R.KR ;8E&94 0?62M#TB\/XZS2:\
M(G4GD%\!C-J%TZ11*<)%K8I015LWJJ+$_>/,=M*D2HZ+\(^@0R3:-**199P5
ME96$L2"G2J*Q0DHZ+URP@M:6*3H4*)6D2A0E Z[.9(<,Q PTL.:'-?WN85*X
M"J83P50,@LDI8:2O@P<//IQ3TBOB<QL"F:Z3B>"2RZT(*'%BW>&_X45M''56
M78$JV[FSP3NNE:R)FC@*<=5DS1=J=PM\[^"F-SH6U>OT-OD3I)#12R0,!9FI
M]#(,<$ NX@?$)DJD2J3$E_!LO0N]NT>1O7 )X008]K"-M!1] G+"N!4XLQ@&
ME[A<V2CTB!Z)=(HF@JS'7B#EJ,FP B<-,5/.I 95-IJ&#-!?02I1'#)$'9WH
M$T"1X80(UF0,( =%9FB(7HU.*GE=X*H0L/U[I@0,\?DYB>[G)%$JD1U*L'?#
MVUR^&\_ES1ED[M[]^_=OZ,FX(O+7AK*U<^>&FKU[PYS'_2#HH<+RZ;45I"3)
M1^5Z4#HM+Q>43JJ*-/0BR MEA VF4+Q@D/ !)YWF!+1M\9B*C3%4([HMIN*0
M'.(2]16Y<&E(6AOR)?(=5:,5J++FTSFD:1&*3#Y(A&\Y1$VI>$ P$3A/0!0)
MJ3*_*9D)Y&D(&G7H;K4QN&HG %'$*N,[L>]')Z#!18ATE>A,@_,Q6RT(I#L#
M^$,2OZH_ X!KOB?!*Z"JK]O].E?"Z'#_'%&TN(:^-\ZRY*SZ[7!13%A)H[*3
M F?TZN?OH]-D EL@/YC7@&1_/IR$T?TTD C7(M__\0'!=8?4L@<@:33I^P\(
M$Z!\87\E*LZW!5>A2V"?:"$T0K,D-S4.=4E"+299VTGYF++UZ]?G)UE @X@'
M'&3L@$I!I4)"CONO0Z3GJ2]FQA GD!U1 @U\KEF6: <;QDZEB@RP22*9MT=?
M#AXF-E*[+,_,10@8 1B_#%HQ7CA$XO#<A%]O\P2TB^O[\!]48 Y0,]"GOQBR
M:M,0VYWP9]&._[A$X97)?QO"VKMX\^E,L%3)=B_/A>>-0T?PXNQTL(Q^=V3@
MZW8D-75J,@@+1M60-0##22J]^C)"2$N<'N$';1[#1>%%.H.A%2B$N"A<#Y19
MDM\)*RJ'<U\>9J.K .J!#7[@DB2/_$!:35,CW"&#MAW"!OGC4AGA;S$@^&L;
M3'\C G9M-K<?]OGI8-!>A_<_[VZ,V^#*H!W;:XMCS\<'NOCEW>1*N %6#'"5
MY-XX'W:0=B)H>1F/$)G2V&_0U*\J+2F@VP%V^GLJ9E3RV\6I>_36S7>?QT;%
M9W@>8"/,.//#E?(\66!V1*WO'QB_X/]S>?F-!3OR&VV '/59^Y(9$VI4J/+E
MSY4),38%&D%7]W\?_?;TU=>IPXYM\>J]UTNI<;8^/LT#\XU*Q73# (P_3_&'
MSYY^[Y&EK:+\ ^1K5/Z'K14$6_#7&VHD*@-27OK3 P@^ LQ^3T]%I$47N# =
M(RM9)L II?@"7^/)"*'=3DIU\-/1QTDW/EOX @&DMO($(.$0JL"J[9 8.T[T
M.).(P'[JSY$)!YP[UM.@2^9J-72V>7%(V_'V0+X7]*7?"56.Z*P-=O[RB>?>
MMNE,L \PE"L\#!D$55#"D.,&V>2^Y8DI>W#LG7!0H8&RUW8U",GGG"\HK\Z)
M]+B6'P=F?#0@M3!$)?XPP #(\9'OKO[>GS\;P9+##+*EOR2C70*:3+8?8CC)
MSS0Z\^*^LN=S88N*O]S*<[?E1U>OA**H"JDS:HHQ&O :KNU75[Z]'@$M-=<'
ML-R*H?TY/ D\)8/R6&'"/[:L*51F!<#J>4URYKP\ATA&)KTA1'Q=A.U9/6G+
M!FYE"6EDS+52G[[.K2ON2A0,X$=C@0BRH2P,1B#RG'%ESZ,W,0J\Q^1H%78%
M>WX\-X=>D)J%WSMSU/558,E(_^$(*+=E^;UUY-<X*"'1-&F)P)4(E'8H82(C
M]XTI?ZQ6._<.=-?T!"=Y)H8I(HCMI7$<CWK^ARMO3)"L087Z_.[W:3BDA1:1
MU<BU@S&;*!917?DW29U+@Y9,X/1$-6%TB&!O08>H4-(Q+.^+(1=,AYUA$[8M
MWYN)G[\_@.3;P-VG- XJ *,I$#=8UB\0ZSU#&IC9PIW0= (84HWNCDC-]O,!
```

```
M63B"@I:"KY@'%PDS.-U=#:IV3.)XZH#0"8-RM$9]!ZJ^2 5)5SBV[#A4IJ'@
MG8\!3!"V2DEM7O+_D,"K="K7Y^^/[Z'PR<D0EXF4UQ1+D;)&EC-/3KB&>3"4
MTW&\BN5\28$HG:'*6JR/"BU;#=E56OJ^?3WR$1#M?D^JP&#0!T%F&KDQFS 2
MX_,:8.B&)G7NVQP_7HW![:@60F+)M79!9_;UI'CG.V"A)A ;<!E\P O,#\@)
MB7Z'](-VC#.MT'=CF$HE; [&IT.=/I5YYNV<&_**"R7=TK;E'8$HU&RDHC&
M!/B@J I,AMJV](1-@*^(=/UXJD$X( UJ 8.X=$D<M56-:1*%(>P3^J-"0YQ,
MHFD.(AZ* =B!R8U%G& *"C>47VOKFZ'A_/D6NOGGFU:$612=:D!$#%JR?EAP
MOST 3PJ3\!C(2>,!:QWU# 7J!MX777E"=>_[:% :B/:Z[8DP2,(@) J1[4XO
M#_+^>8 $X1DL -$\@(.C"M,1&RP%^B\J6+52S7\ ?L*=]4%1+*]/@^7=<9HR
MC'_?44<@I/=Y3C6<_R6@\( 2#G40&C-0S%<_IS&;JR&]^TGH*).EHF)V!X%:
M4*![V(Y$0F@IH"E@X +1@DYDA,J@KJ%IK2I0GH?Z!O?OE\?1-7#\>30TM,/R
MO]?IUR?XH%*@H&B !ZPQT'L4!;)_?^(+3L8,E+ "5P[P1"!7EA+K>X*3.XM,
MHPD9?1'BHUA__W]O#)22\($1!!/H8F$#&P'*(.$"<@P%D(PO!P',3F( &3A"
M8>%Y"@%J&RH($AF06:;IHV%D0R,3 OB_'1H<[SX0'$-YFBHT9"EJM;XT.1*Q
M,W$RW='XX("F"*502(: )*U9LKW4"$3A>0\*<E"(= 9@I^( CL@ #@120A!#
MQ0B D)3*8D7AB!8H .:# 1@;UZ]?CSY4;J2O%;(J50<$@2(D:'*=#QZPR?C@
M@#(:*I'YWR9 &Q43.)+0?>)HHGC*5/\K7VL/]QM24"R3QM<EF4 NY "Q$53U
MOUGE =.S/Q.37A/(M8$$GLQ'8L4 QB@3.#2$05(DP/[&E9Y(*B+IB(0N,&0E
MTC.K'4ZO$5*NJA! J8SOXRN0D:\0\\0,]7&5(J'+3<#F$CD+-C>XR/C :PK
M*:(>0#6H"4*1:H0%Y!%%+$\XL/ %I"RAJ,U9XE.N)P72ZP1\#P+I@RH?MB/!
M[HT$&Q:F"2XOE)LY86YF8$^]F\I;_+8!I8*9AQZ$',H^WH9$*56'4(]0U2<F
M2X]T@Y6;(NCQ)81M4VR/_/Q%JEYEE/\&"NN-$PQ,HCC-*6Y8 4QP@X<KD3*%
MU>OT"#*%"= 0(OX#PDM)DJ]9G\1P1YXA \:#09"#+J:\_HT*VX-+BV^_=F!Y
MVHGCZMO#U<$1BBYA,+T55W-1!M/]T#7V/EAA2!M!;[5)8?2RI:6;RU;E&?H/
M2W^+$/M?%S_PHLRE/55^N/HKG% ",8:Z%^AOO'!8-V*%OW!_^@ C_P=(U^T+
MO@=8/Z'^]P3RUN=,7B]LSU3Z9X]7BLM_,&1_@&X,,)ZI=M*ZL6([.[S(D'S[
MZN*OG7:V)LS2DV./;FVG?FT@MXI_@Q!4<'9?CDU,X NB'S#CPI +-106-E[R
M8D<+:"L2H7<LLC(]1@;!NX#?[+XI@ERZ,\7DW:F"] ?P=JG7W!$"EUA)#&F#
M$OZU)N(!\ZI[_6 _#Z0)$$J6?E\^"VY<C]%I\A]S=I1(OQ_+F;XZE_$[6;I^
MP0E!>7$4SM8N4$_CD ?_3?'.3)F@=K^.;="!N[MA[?=..;<GB7L$\U5^JG5S
M$L<"L"K8+;Z'4)=DZVM_+T+\G^.'L.NW%&GA\ \HFB"MV8F2D+P\@,I WARA
MQ% WD#AA\7G$Z\=#Z*3HL,D)/8)\Z$^-IU=9O;MC+?#YM3G7Z\2[^93+C:.L
M>@V]HK@Z!*Q@7192>Z*2M7W>'\V]0"U VAQ=;X)H^PXOF+MTM?P&7)(&8XH&
M!O8 $EVX#$+'.D07)!6&0+M"-ZEFDS_\'P"<*)J0^?K]"BZA0'G_@)!G?BY'
MVE#C8/P &?S$&_JO]_24V.R,!DESLO+_=U2NP.7MVKM=<5F 5()N6%MQHC/-
M#@WN "4GK9$:2*'<757&-X%S%#.L-Z:0#(FJ21IFZ?U%P$P +MO!/A=:LOX
M1Q-K=90<' T(CF#.6AKB F-_2)WO!8X("87Z 15#SJ?4LJ;WTW^78T!PUN/-
MJZ?_#Z)XWY.KND558-4QZ6$;$^XFA-*MM^?L7(Y(IA\^6)G*>Z+8^(!2HY\C
M7;($N([X(*JR25PU@_V# TCUP _2+$4XDO@&". 9@<4<0<DJN%7=Z:$=^C@"
M^A;@L.4#5[!WB  "BMZJY'2:[/GW@$U)3>,JIBT0D'A++=<C-H'A>50'#S2
M!$4-&XF:8D7\0!^+0C8G"#KN]_()-(D-I9DDFA+L#K;[:8;U55F,H-3T,PAB
M\+Z(2B0&8FA(AR4-S0:4P(347"[$(+F4;N1E42#K6B6%(268D 1E'>E=2COT
M,#B >TB,Q!$P&&"3:\Z4P)76K>KY08B[!R992HP_0$'Z<]OF8E4-@W1')$\#
M(/SO1N=ZUL3ZUB8QJUH&&^BR!#;(IYG>@DWXA/ JTSM-'3DE1M6B%5U7C?W%
M1V>'[2TM:B8#H##%H !S4$2B<((%@S1 B!\E+[:@8X<FY:!?@<3PP'YD#5*K
M%(T&(&[B6E#*9@-T$U%U%#0-0&SRO:L.N0\5,70=&9CJ4"2#I0]:E?:_<2-0
ME<34Q-1#"6F?4,&*#+R!48$68;"B?V)_4A,3F,LT ^,)3,$*=C(=J""I[H$"
M5@SD3"36^,#= NTP[/]?YA!%MWQYJ(&4$IJ=\[FH%")*-5VN>Z5D(PV&&HI^
M4$R0(,$"%!.R:GR U% 9C30:^2')("!NY@*A:Y- ED:WQZMUJ(YWT#"_[!C!
MBH(+S$Q8P)>._ IY&1!<0,&Q/@F/W;>KHZ!^%J^1\1IQE:63,G3#!CN%#Z@%
M)&H2Z(AP"QP@:N0FP@]F OZGYM#E4.5RYJ@JA3+KNV.06*--0GP#R9#(4ID_
```

```
MODJCHB;9(()&E4AR"G43.HVX+83([?:9_R4*+WAJR*U#(8TBH4Y'%0HM0Z&Y
M4 FO:)@+Q@(%Y)(HJK"ZA)6,+MW@R)J0'4Q)&M9'/"2F*!*A#9(J(A5AN5X&
M@1Q0Y.<@J$:Y)FG%ZBHG<+"_360$M\0+$Z"(P/2G-Y? STG5<NR@,C%G! @)
M#U"A$$>A[EB@/ZV,%#XKD%:8@#YB_?I6#=S?)_,^Y:!G1?V;'89,@(;["R!$
MY*8((K/\>1$9W \;D39B @APPG*AAHKY[UN3T;((5 @CT=&')"%X@5ZC7;1^
M?MLF?\$-U@7:C*H2U]O#()((O<DN<!MT:<E=?K>A$JMI7:6:\7D5U!OS@Q34
M'L&' Y@A@2LL%]9*IRWH[3>H/,4@[*'K[#&J4Y$'BC(.@ EA  ]#'.=S%+(,
MQ.[&[E1LG3Q$DV@RMS0*LA\1VNF"X/XV0\D;([I&UYG X!S_X<T0#:T'5!U8
MIC>&]CP3'.DD#?\^#^H(81*%08@-X@PE/+4\@6%=@6M[?O+Q#$%0L(YYPP?D
MG#9RK++X48#5<69X## =+N?;AJ1(%+WP4-DP'.T<C,_9AUQ9\^O?MTXF. .Y
M [4%WN5*E"0>@B5LA0CV^8IZR*Y"/63#]-:E68![ *L(-O'>MEN;[FPY<N/&
MC 3?,>3(D1\_CBQY$N%W?NSX\>3%G0U'ODRY<F7/BC-CSFP(_#/W8BI/4VBW
M(U9'6#$K&27C1HK"*XOB]*D4*=DE4>$W,LQYQ^;%(^8!])_ @<_]0P&Z[%:G
M3Y>Z['G2I<D7(52'!3&^3S >38^G7_PW[MBC.X^./#FQ9$,8P8>D/( %%#A
M%@RXT.!#+AQY<_+<KT5[GEUZ=>O3MT<-,;>>O3KX[6(ID,<))!G+_&"YII*\
M>"&&H*RILI I^9(:]&YW-RKE!Q>"6.![&//.X*AZF\*33]"&7N>+*FB7QS=V
MD%1%2'GQ;IA1 JH>\F()Z_Y ^!5%7B@0(OT.%]PO<\;L>3-HC1ANG4%]-/Y(
MFTFC6OECC=8KF#OVLVWCY!'GV3>@^^YI"LM]6Y=MWKE[^?J)'!4.W\W[-JX:
M.W_:.:T?]FG0F3:J/^2)4I'6M\((+*LG97F_-'AT.=#FV+TMR]>7]YY=6W3H
MR9@1/R[4XA^,B' ?%CRH-  Q6M%Y4)E'\GKERY]+[P:.G+AN8 ^X51V%3Y=^
M7=K555BV[UPK8>#):#.TE7AW:]@8)K73\H8UB3&IMZ.)#I\N D$7DLM)-?ES
M_VAZ/RBBE/RE?+?B=D@N$/"%Y(HAQ3:SX;)C<Y".H<:^$R=/'IN5$DK7CSY%
MVI3ITB80\*UBW=I5:]0L?L1NY]ZUB_=OGM+T*P !\J:ER^YK=R]>O'OZ_&_?
MOODL7)FLV;%CR88=&S9,G'\K&,4;_J,^NLACLN@_MRYO+0M1G('^@?'AB(JZ
MND"I*;W=>;(G^ET_?EM;OB"O$O_L62S^UK7N'9$/!=_UZ=6F2Y?N#%KSYLN6
M+U_NC-ES)458/M]9-*DK7['L^M7J<O+E'I!W;U](_MV[<D_V B_W:==2\RCS
M=^_.M>;Q;E\>^,(RF.2<D3,3H"B,BXR[VA4I%?QG0F%DED?XRI\_N[P\; E?
M'R^D1B]+3%]C!MFT@I-L*V!(KD##B;9&0AVABV >87Z=.C!"N!&8'^C/GS_*
M?+J56[3]M>Q9MF_EUL'-=@KP*%LM&B]O+X?@+8&O-6N6P+]ZA8#T5"EM/C>_
M4R5,8:)#WGQ+X9LXRH)QU/EJ$*]4T9?N /=[ZU+C':!)(K?3FIS8EE\DF'<\
MHDQ:,0I^<"'LV*YHAL;C;][\V0Y&.I^E+7ER7%#>R)J\?^>FI<@IE@CNW\.H
MO&3E:UAL/<[9JS?[E5;YZ\R#(X]^O;GNVL*8L$L'AH/_Z_.'UR+\Z\_X_KV)
M95B96/U>#^EI,>%I9#X*^VRJ-?G9GSF0+6">OLT*$LP+Z\ZMS"VF=X202%[<
M;=$A"KC%EM>: -#1#[PDIZ1;#^\(&ND- 8Y'MEBY4O.7J,"_ 0)^Z,Z7*IJ@
MDW11Q??C@_-!+GNYZ12V'V(W <%WFM0((!^UZ1'"VVRD/VV<=X]>L8W*^&7-
M'!.X$.#=W&:_98O1,GR6*4=V #G\P!():.*%"R% _/8LK5F<OY[#0EH:Z0S+
M6R>$4AP_?CPLCN6)')^[2NJ%H^NC$$^@'D>4<O5JH5$7_/#AQ9,[AM+RA,2Y
MVP1(\VETS83)B_>NO;JT]Z+RID:"1Q\-H2M"NUJ6IX454Q#<]G5J/'B">C3!
MM2XUF4:?CH3D<T DGS^?3HG#\^?92-  2EJR[A8S "CX052D-4]*!K900Z"O
M%T$0Q=)ON8+T)<7J=_[N!7?*E"I9N.#Z(Y;A1U2Y+!< ,I;Y"/-*N7?@6>J!
M4F/9,#(AW;%C!PP<,U!I: ?)"=8\&6424QA 18<61WPZV22^-].!^Q,1N%_A
M4!&6HKJL,G25)34)"DYG'AKQJ,1%)DB6IV(Y?OQX 1W481+M<A@J<PIC0O.A
M]OET20&%Q^-,AL4]3]UM3CQIKS@PKPL2_<Y)AB1-9QE11,P/IT)WA3@2U3#R
M\)Q@+V/__XY$-+'[L<^P@>,.@:.4\\</=01CQ(CACA *U!<O? YOU& 5T'TH
M$C4CRJE"%++P,FD-$D1X@)='?#RAB* ;%70G(G#D $0)[I? @V 4^E-1$I86
M64VR]D10L9$!$@\D4"/  B82R [#\_0R0G)A>Y.#KJ,^*&5&N)WNF% 3A+9C
M=Z1!M",& 0JID*>9P(@$&_@TPB)A 3X2#, CL0#'0?)H?RR.[R5(B-[T@T'5
M5>?RA!>'?@CZBN $<A2P QP.".!+0@9(5,*3L5=O/7O[?_1;H8/RQ"\&R?3I
M20?2"U31X5_,VC0#WA&#J4[A]^Y4!3'[A3<SQCOANI1,C/9AGD*5XO"S0/SE
M'//]_@$Z,,B%]- L=$IZ/$Z04GSP8E2?$6H@U^OPF'T[PD"!W@ '^ %ZM-]9
MCKI87K^^?.O/S^--':%Y5^BS'2+P1_X4._0#O=I!B$6#K7%I3 1B/-=;R1 6
```

```
M##W;_56V+" ":K[#<XL&6T\F,G@X=M.MVV-(QO+8GV+].A9MV[=SZ];IZ SV
M:M2D0XK.-%W$@$Z[M., EC,]%5I2,EJ3/C[\]RZK=7/VS5-;EAS9\>+#A0DM
M=*"7A)HE6]X\673MS;I7A:RM]Z?^/#9S9G?7U<U/[Y]QY+P[M!/ E+2D?)3T
MWDR%3(RS"-)W*@FP?P('?!YE!/!_ N>]=M\4[JQ$9M8K//X7)=R  ]3>ZN\R
MWLB/=>QHWL&VZHTT">"$%@<*NE#F@-*F+PV42[T,YXZN/QW<$UN.E/L1E2+O
M#@9417F] 9Z<H%PB)C^S9U A:&NC1S#*<[T64<1[_I@ZGMET[-08:^L1@AND
MR1*@X)(F-^R\I$Y&@C?R!)KR@MOTA;VHKY/&/>*!3;F?0#VDR9,J%KSD9FJS
MN?G0H4_?KLVRLF]R-/WTJ9IR#S;JU*M3LV+5\C<=UHQ<.>T:U5X#JNF;_7S2
M>5#INW7CA)+7>AV:,RD7D#%9LF33J-)GRY;]ZG5KEH9Q6+9>F*A?F%(^]8@9
M7XHLZ&[8G2@AS Z2AP/C5;].7?ISI[6\9]JW.2=:#>" =#%P!2[X]*C7@0<F
M7!A1-<:3/)E2,1VJ6"[F(RY<%[Q^O-A>9V2D<NS2O:=W,YK*Y>OKDX.*0"9Z
MSJ@DZ3E2D' UL-\H\R6'%M0(?&BL$$$ &B(%@J,,LJ70HC=3FB1!A3)(D!]!
MAC!]@1%+TLQA24YR-;^*I1#YS"6^YI0U5$XC5S9K=D[B7L-7G 8T"ER"KP;Q
MU725V[I]._=/Z> SP1AFKH+]/I\D5 5NKEI*D\G8:1.4<5&D3Z4>S?F";2\B
MJ&:W_]@@#HV %,UF[-.?+]OEBQLOMBT?C^;/-_=]>_9K5KP=^O.E<6'$DH+Q
MRYL[^?O1I5%1XZE5J:)=YP;^N_8IECSHSZA&Z=:;*T^R<FT94[3DC@ENSJ ]
MJ3YJ5N1X+DY\.3#EARU[5C,I^D=O-GW+F6;^G+HUNDDOOR?I<S"\<#';BI)"
M0/1QKR?."\0_KV^C!Y#HAMWY#<;78RL"W72+%"%)HJ%X@HQD B=^B1NPIE_.
MI&GCKR7;1Y,N?6+NJ5[M@BF'Y7**5]5"#A9CV%,NR4HN\DF%#L&DFU!A$R4-
MXU2V^XMPA[V22TEFHT%_[HS!R=;9<T=!E4,5ZA\M^N.4]?@[ C_G3!6:6CR%
M2Y^8^R<L4 -&D50Q:L9D<CM-5D<AN->3SQY].7-G?,[*"C',KMU[6*7,7-FG
M,#)G'%KE*:*7.P-&&>+(C2,+Q!+Q8FZEK'+M6).)'Y[Z%!/V6F0E@9PG#FQ:
MEN;RCD95/#S8XO>^M9"_[;N8\GZ&!<VNO!V<(2F#]#K"E74;;7<F]O>@5F>2
MI"H5-&E.@EP \GQ4$GHPK> "KB[1!D0,(Q1PM:OA?U B)Y ?@0C,0J\_HZ9\
M\\8D7O,F#T<@H.BB29-L U'D9/&9+TC1+0*?98M8,@W0OVG3QB^6V4-*^R8-
M5=(Y](?'/HEBF.2,1$P2IE^ZN":O"/P6R$3:](IN5"P5 .] Z$90SQ231P5=
M@HMBZJPV] JN.*E&'-PQC*R1,@YNIFO;I#)U[!!$72.:19JVT95 DRQ:[([3
M)>LBO^PC^_DSF:R6W)O5E/TWE.-J.YF=*]-<)%D<L17XX.X$/9*,Q^B6L.)
M%VAQR*(R/.^?0*$+ RF_"77:_.!H;*"&0E24U)#\[X*E4Z"U_H9%0IW (462
M-(G"VS+W+U\TV,+A"9*FD(KT2(1*J!CXXP?/+J&,HB4J; @/PQ4^)^#S?1$Y
M2"63,P0NR.= 816B"[E/L.:(#+YDR$0@)PEYZ2 \'4HD]*Y!YG0G*ETIU-:I
M!N5Y.!/CFQ%0@5(95#9"=+Q8H&=S@;Y]VT+>&\1@<9SH<H$K@1<+3$C%Y2:F
ML?%#%E>>V=%(^%W@4 M!, C^9X>@,.]XD4""*F;$\28JVI+1C&!_1PT&@-50
M078  X43-"5&<-8)1_(#8R0"C4>.!'&8&G:::(,&Y"CZ4F &HR,JA#KC#'BT
M^B07&!$0$=P/(5 C?G!RT4)&"EF!JB9MBQ&^X,)34$3C +#QO\!A?S@BE4@4
MK%0O$#SN[P.)4%3TU0AU!$""""X@;SRLK</MV;1V%)E>D22)31*A(Y(3GN0?*
MR)8W&/1%HBM%U(SGQR/ P?DH%-!=1I>& 9 D"!?Y'PB"O"IJ5V)UZ*BB2C,(
M7(#:'.!)II(,0BFP0 %D'G"DBSS4)T,U2-3'R0="V&EZ+M AJ 4YP7]+)$UA
M(6 F!Y<M*$$H.03+GD# @0XJO61<L$#_N&+0%7_T+"?<4DO<VKRL"FPO5Z[L
MYM7W9\YLQ*8\2S.8:J'3<:9'63[]>H!!4YD!"+AH*J%=%3Q+:STTZLJ& ^H(
M$H/Q]M3+^QDZ0TA:E(R5$.$B18!"=T!SI_!1P KR"/(4DBN80$<"ML]L0!W^
M\61$>B)+/,[7TH&E!>YIAA>X)"SN'FX,S.@R'&C2U""5B-VH0$<.Z TT=L3
M:2IUP><OB,"OL,#WOI'/,Z4PWT&)0SETVC)&#@RLI&)P;*VL++H5*N'\S\ZH
M _4"V"<@$HKN@OL/X,X$^T$>J1[_8 :-_<H!JG+U]-WX,"][>&_K^R<KS=*L
M%EM#3?T83BB+<GOY]L"RZY&D"^:#EP\^%&B@RSH!9N+R>=9(+^1$<D:0A;IA
M!R?K$/9<YZGEM-_\<O<#8C65$P^NB[\'%5JXY0O-X31N*&%6I<!) "ZA+L1+
M/LXB(HMQM2,5^/G]V3R-$':0,!_)WUV'XVXCEKMUW1R=D/ V"*MK+"=,+DQK
MX)]^GI:F8(UNU,S&UZ*3 IZV?U!)[& B"&1O089,$E1'ARY.6/%D#<>KUHS.
M"""04_<>[V_.(#5MX-V\B'8 :KL4;%N4T..(7C:0=C0'(7;-27F &!;=P.61
M\BT(EMR0!M"W?3'FT9(#XU&6HD.6(/K.D4T+3QOHA]_6\28W<787 7/^__F
MGOO7 TCS0Z(E]L^'1 @HGA^ G"JIKV)51"#1I$<OLB;+CQAV69 Z^FMUK7 8
```

```
MQ-]@0X01!DRNR""&\(,%@AU0"YVFTDHO9+$ #MS^&@B9X ?(OJ>>TY/@ 7;^
M@0Q-L@RY$%_%QN*6MM!\4KO [Q:L3>O?1+;2$%*;6>!Q6FQJ<B^^(0B:C'4%
M1W ?J(#(\*ZO\0"U5XXU/=Z);!!&XA24SOV&_O;]@9ZA^]@=\,X0K$JS7X"$
M567A$!<,8 *#FC>!&#@_>#5'R+$=(#?Q ]5;"T2:L@0?802=6L.<G! E3$?X
MZ?C03N=ABHSJ[\BL%NF\7@2/F&9P/C3:^.JJ0O T)!EK'3\]31FX1B"*9]$#
MDM9%;/,48)TLQ\I]*$5)%% X0A3M!Z"3PY]"2=$.(X6OMZ^1]X"EXP'9_@ 4
M(658M;0&HY1[ +&3W.GZ'50- 7<\."\\\W)W)H\+W.2&T>=HX 3GN#Z^GTU9
MM/W5/60(/& *I">H6*2X+7"W>.7W\^1^]*?P!]$.Q:D'W%<'1CE; -\#VE@H
MA>8$4F!\@.LABBT)#EV89 ,A>R>%%IA,J:+"J[I"U -T@6CIN!T"(V?-Z4!&
M1X$3FB7#-;@\KVV00M-J07AELQ$:X&*@F\4(" (C^.*P@QHP!ZCQ5&AY$ 3(
M"-#40!.63$X48D!!8$TV1P"#[CX'$#%A '<;":YF/A\=ZW4S@6:KB9(+Q._G
MM8%!) S3"QS'Q\>ZE"Z"?7P_((I@],&D <$,'1!QM*V"53-$\+60*=0#(1X:
M64)56BBTO"9I1?CA@Y*O2?  , JX_GX- J%1--5)B(: 6:\OP*K3LJKGOH[!
MH,$)I OL**&SO-O1N;,&M&UDM,A:)P!U;:"(5+V51AS0P->NY4H-K#A1JTJ>
M&8&MH@+2_F 7@FE^Q>E71</[ @Q8>&@X0 J.K@GC(V,1V91$(']!)H&<*,>7
M-V6"Z]>7EVENETC<7EY4X/W]Y'2T-M$M2IV'Z(#+((;I!$P+& "CW6PPH#\]
M EF\$38@3E>=3-M EHX(#"- -0:0AT+.$PZ"G"ATSPD,Y*"A=!M!2D<D"L)5
MS^-K/(_-0PV-Y@;4IL#\$#1H7-R/W(PP <6E!IH3)%4[". 5ZA"L(^X3<SW0
M'N%2V/^C._[7TR1Y <BF=O)]0.VD\W0QCZ!P5,-]3_2($.*Z&=WL")QNCI"#
M?OEW/]D!T 8Y6YF'K%7*"1":_85.4@R#,X*T!:UD4+E(VOLK9/1@"XRE86@#
M?)]\1:.FGF%^;PJZ41#*A!M < )K J!0O#4H_4;T*JNN6.@#M4J[L()?R(YV
MXG3P1KV*N(4.4X**QH7")5)$J^Q]"CB[ _,HK&+[]^@:ZL 1/D"I2G0#XH\1
MJ*C('%+DD"(DI4%O%": )D*4V@9!07\)$V '^%7^%R7# 0/T"N)'KI]/3R-*
MF4&\0".* B&EJ>A#9GG RNYB.N>3D"#<]^?.E/39(((\$B:@&OOCGDAM!1.%
M'9"S"<A5A%;)E[']>5($NL.ZWB7+=UP@@,0&[*#LH*[[ :E1:MM;%6TX$C<)
M(V%,'CC 1$2,RQ9N("XS?OKOOGB>/P]B<3NR(TP("QA-3/L+8]$9,A DIO_Q
MY]RB$?#V(+<:?;C>6P?V:VQ)!EA)66L[V;^GM[O:I%8E!$Q?G^VSMQW^Y4@"
M"B+)N[=5F2OTY0B%EVG#?4&S+Y@S;=$-@#XA3IP1#.;RUX^3<(IP+#[=RU7"
MYW5P/#3.N/_[W. $,#_04M15O>#AX@8SL/A!"::6KXSG#E2HAW:DP,;?,ZH+
M$S589G187]-WX\8#1WQPN5A*15V[YC]/>WI,I!]*_@3H$3- F3'LFOH5$A!L
M $H0^!_TF74%S(K37^Q9TO^Z;03%,M^SGN!@E(= 0P[]M4D#3TASXKN4T9[^
MONEBA(W99QBELG0%-17@:8M!)EV;DP&O*_^6( NNXX!69)T^>WF KU!+ P0*
M  0 !@#Z:,<:33MY=.T!  !4 P  !P   '!W;2YD+G-*!QB= _T= QT(!4T&
M""T'% 4(!@@'%@@&!2@=!"4#!18#%@0%) @$ P0%!@@&!1@-%_V=_OYN!/[^
M+A<>!PXG!1X&'@8''@=^!?[N#P$" P(# @0#- 8;!;O\_#P&  42]_?WMU&C
MIO_T%^$DG7=C^]N=PAW;Y!<3,OMA&QTZ[%O6*+5:E'4%+>FA$L]^<;?;9QA?
M?M_]YF[(CZ6WI(8)%^CKISE7F<OQUO]FZ))>?5GM__UPQK8)LJD7Y[_G0C]=
MS*XIJZZ>\;2_F4KS"][T3+6[^[8O;CCE^Y;?['9^]5+:FO:1)2A@A94UV"7%
M58)P!0;]"_,9T>WFHRL.;RZLL,(J$5EAF<Z;ZSVFE$1I$HQ$+A91.3*JTY9,
M-28_[?N&42:LE\U9I2V_AS;UTKH@^6<IU\H8M3]]Y\;R>X)+\SB@UCZ!Z+DT
MTF_NT.F?XO;3[C+5[!Q_7^17.>W^33O' CNUR-E<IEH>#-PUL?<WE1OX2][Q
MI_\^<:8:[>2\#^PEGJ1$C;B6J3K=)0MB7V%-GI9F'/J/F-+-<?C'^UC_)#XV
M+K"&?@T=\7REG]"GO32,U^$/6]ND'CYFF94&M\CX8-)RH3=P-KPW @^;^]IJ
MY4)#IU27U<J%F_($XCR!P2]CTP]FDX3"!&93#4L!4$L#! H !  & /9HQQIR
MY4!WL@(  (<%   (    <'=M,BYC+G-,!QB=!/T= QT(!BT7)@@6% 46" 8(
M!@@6*!T$!10#)@,&" 0&) 8%! ,%%A08#1C]_3W^S@7^_@X(#A<>!PX'#@<&
M'@8>!@<>!WX&_NX. 1)#! ,'%0P$/ :L_?T%#0 " P8#!@<&%P;W]_='D2,7
M[ 4_PDVY_S7=*]_$-YST'Q,J=H=M=)BP?]URJM6<;0--Z.3;_OA7;F\__%+[
MVG_B^M^51LKS?O(;YJ?\Z<\R_KYTNM+YX$]K_E3]<<FO_"S/FZK&+Y:?Y.>?
MJ73J/]?S^-K<_P.I?_O.N4,*Q25^2>S&>TK3^WTJ:-HW]\/$MF,]\VO.FODC
M+5U7[IN(QM$;\IWYF_MR?^)25]Z/>W]>AM^-2_TS0JC<\UNG^UJ;]OSDFY+\
```

```
M$TH+W->Y=/Y[L2^G_,)U@\A[U]_A%(,N%J8NN[IC=LO8=',:VKS]U)!)W?<[
MW+,7QVGY(3(?E59^0OX_OK_S^^M>;,_R"Z>L+8N2*U%I\=7=W(+Q#JYMV6N&
MJP7E"@6[)RYW0%>9NJY)=7-CA2NL%B@K/-.M(/XF%4J,5AS\;LG!47(&)6,L
MA:D$XTQ4K:+BB5JRUJBT&" W_2 4(+5LJ:I"DRU_E]8+IVV5ZHN5;65"[5[0
MUL/\O^#2//2HM?? D=P$MA=?7]8_ ZF?I<:8SM)$I>4V1-9*.%[%5JKU-,K7
M<E[;,B+2>/YK?E[Y40T73HU*BR:SW<7).&'9*B_AI[=!WH[#CK.CTHC:-39<
M$%_##[">1IRJXKSLHBKF:OKH_9S8_4@(25X3D1-,F9Z2=Z4;?_K,XI-.6Y.D
MK)E>@VSC*N$3\@V;%SY/*\,SSIX^]</S\+YJ[EC'ULE<997\U2&.*3PMAW-X
M;T!P5[.-YY3*BONREKWK1F6EXCQZ."<QO(XQ.YK1DA)'-&/,] 502P,$"@ $
M  8 ^FC'&LJ@5',0 @  _@,   @   !P=VTX+F0N<T<^!WX#_@X' QX'!DX&
M!RX&) <&!P8'!@<&!2<> P85 R8#%@0&) <D%@<%!A<.%_Y^__^/!/__+Q<?
M!P\G!A\&'Q8?!G\%_^\/ 2(3! ,$ R4,!WP&7/W]'0H $@87!@<&]_?W9]&B
M%_R"%VQG['=*_>EVRAF;[!<?D_!A&Q<P\%O6_'G.7U?0DAZ;OMN7=+IV#^-+
M;D_[;C<DC^6W1 P4)-3KIYN8=2NIUO]ZZ#)>X;+:__MACTT=KB[,O+^]*?UT
M,+OFK+K9XVY^/97OBVIZL=J=]C0O:=B5\Y9?M\U]A>6\->\C:WC@P.H:><EQ
MU;"N8(2?<O>(+FY;5QW:7%FA+%8-S@I5Z!:N]@A3(J50) E?,C-1.1=5\I9B
M-3YY:L\;1AF?]'PQ9Q5_R\^F=6%Y79#^2CE6)M3^@C9QK+PGN#2/ VKM'5+/
MD1"_/D.G?XZ%?>V(I^TL\:LFW-3S<E_6;MI?-[=$+A1^FYM8K%8&%6>5W/XZ
MZX1YV6WJ[K^8S\5JG)QW'_JE[XQT'Y_G8E6N+@E$[BZE('M:S/8X]!PAS=)>
MET\BT.09989E4Q;W>![_7UW%/,"9P;]T9=;R,!Z(.D,$G.&^Q$V>@0ATV!"A
M$D0R?US_>@-Y*WBAM8XGK#C*HI4Q<26EUW(;U!L&-Y#]>55QX:'3<W1955RJ
M*Q.=>QK#&YCS:<R:EC+1F.-F*E!+ 0(* PH    !  -IQQIT41>R90   &L
M   ,              "D@0    !D:6=I+20Q,C P+F]02P$""@,*    !@#\
M:,<:E85SFV$!  !R @  #               I(&/    9&EG:2TD,3(P,"YS
M4$L! @H#"@ $  8 ^6C'&FM^*+ + @  > ,   L              *2!&@(
M '!W;2TD,3(P,"YS4$L! @H#"@ $  8 _&C'&O9"0BNQ @  \ 8   P
M         *2!3@0  '!W;2TR8VEA+F0N<U!+ 0(* PH !  & /=HQQKFO<<O
M:P(  &D%   +              "D@2D'  !P=VTM8VEA+F8N<U!+ 0(* PH
M!  & /=HQQJHW6A*AP(  .0%   +              "D@;T)  !P=VTM8VEA
M+F<N<U!+ 0(* PH  @ &  IIQQK396D>>SL   )0   '              "D
M@6T,  !P=VTM:FUJ4$L! @H#"@ $  8 ^FC'&DT[>73M 0  5 ,   <
M         *2!#4@  '!W;2YD+G-02P$""@,*  0 !@#V:,<:<N5 =[("  "'
M!0  "               I($?2@  <'=M,BYC+G-02P$""@,*  0 !@#Z:,<:
MRJ!4<Q "  #^ P  "              I('W3   <'=M."YD+G-02P4&
/  H "@ O @  +4\

end


.......
....
..
.                                  C=H #20

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::

cu next time!

-S
.......
....
```

```
..
.                                        - fin -
```