PCDAC

By: Michael Herring

The Data Translation DT2801-series IBM-PC-compatible analog & digital i/o boards provide an inexpensive analog i/o capacity for the Xerox 1108 with Extended Processor Option (CPE) and BusMaster interface option. These boards provide for analog-to-digital output (A/D), digital-to-analog input (D/A), and digital input/output.

DT2801 boards can be mounted in an IBM PC expansion chassis attached to the 1108's BusMaster interface. If the direct memory access capacities of the DT2801 board are to be used, a PC-compatible memory board must also be mounted in the expansion chassis.

The PCDAC user package provides a convenient interface between Interlisp-D and one or more DT2801-series boards. The PCDAC package provides a functional-level interface, so that the user need not be concerned with details of the board's command sequences such as command register formats, handshaking, and error diagnosis. The user may instead concentrate on the design of the application, and on data manipulation. The PCDAC package also includes system diagnostic aids.

The PCDAC user package uses the BUSMASTER library packages.

THE DT2801 SERIES ANALOG AND DIGITAL I/O BOARDS

This sketch is intended to be helpful to the casual reader of this document, and to establish context within this document. It is not guaranteed in any way. For details on the DT2801 series boards please consult a current Data Translation DT2801 Series User Manual.

The DT2801 series boards are identical except in the resolution of their on-board sampling clock and in details of their A/D subsystems. The A/D subsystems vary in resolution, maximum throughput, choice of gains, whether single-ended inputs are possible, whether unipolar inputs are directly supported, and whether bipolar inputs are encoded as offset or twos-complement.

I will sketch the capabilities of the A/D, D/A, and digital i/o subsystems separately. Where the capabilities vary between boards of the series, I give the lower capacity first and the higher capacity or additional alternative in square brackets.

A/D

There are 8 double-ended input channels [or unipolar, or as 16 single-ended], 10V full-scale, with 12[16]-bit resolution, 4 programmable gain settings from 1 to 8[500], and maximum throughput of 100Hz [27.5KHz]. Sampling can be triggered by software or by an externally-supplied trigger signal, or clocked either by an externally-supplied clock pulse or by an on-board clock with 2.5[1.25] usec resolution and sampling rate from 3[6]Hz to maximum throughput rate.

Input from multiple channels cannot be sampled truly simultaneously. Input is either done one point at a time under direct control of the program, or in block conversions under the control of the sampling clock. In block conversions any consecutive range of input channels can be sampled, but they are sampled round-robin, one per clock. During block input conversions, the board cannot do anything else; in particular it cannot change the output levels on the D/A channels or digital output ports, nor read the digital input ports.

Block input conversions can deposit their data directly into RAM memory on the expansion chassis, by direct memory access (dma). A dma buffer must reside entirely within one 64KB memory page. It can be arranged that data be deposited continuously into a circular buffer. Data in the expansion-chassis memory can be transferred to or from the 1108's memory while the block conversion is taking place. See the documentation for the BUSMASTER library package for a details on expansion-chassis dma and on 1108 access to expansion-chassis memory.

D/A

There are two 12-bit output channels, which can be used separately or with their outputs latched simultaneously. The output ranges of the two channels can be set separately (by jumpers) as up to 10 volts, unipolar or bipolar. Triggering and clock rates are as for the A/D subsystem.

As with A/D input, D/A output is done either one point at a time under direct control of the program, or in block conversions under the control of the sampling clock. However, in either case, either one channel is output or both channels are output, latching simultaneously. During block input conversions, the board cannot do anything else: in particular it cannot change the output levels on the digital output ports, nor read the digital input ports or the A/D channels.

Dma is handled similar to the A/D subsystem.

Digital I/O

There are two 8-bit digital i/o ports, each of which can be programmed (independently) to serve as an input port or as an output port. They can be used together as one 16-bit port. Digital i/o, and the setting of the directions of the ports, can be synchronized to an externally-supplied trigger signal.

The base i/o addresses for the board's control registers, and the dma channel to be used by the board, are fully jumper-selectable.

A screw-terminal i/o panel is available from Data Translation.

USER FUNCTIONS

As documented in the DT2801 series User Manual, certain commands can be requested to delay their action until the board receives an externally-supplied trigger signal. Those PCDAC worker functions which encapsulate such commands have an optional EXTTRIGGER? argument which, if specified non-NIL, requests this feature. This EXTTRIGGER? argument is not described separately under each function.

Miscellaneous Functions

(PCDAC.SETCLOCK NTICKS)

Sets the rate of the board's internal clock for subsequent PCDAC.STARTREADA/D and PCDAC.STARTWRITED/A function calls. This function encapsulates the DT2801 series boards' command sequence Set Internal Clock Period. NTICKS is the length of the desired clock period in 2.5 microsecond ticks (1.25 microseconds on a DT2801-A).

(PCDAC.SETUPDMA PAGE ADDRESS NPOINTS WRITETOMEM? AUTOINIT?)

Sets up a single direct memory access operation in the usual way: one would typically call it once before each call to PCDAC.STARTREADA/D or PCDAC.STARTWRITED/A with DMA? non-NIL.

PCDAC.SETUPDMA masks the dma channel PCDAC.DMACHANNEL, loads that channel's mode, address, and transfer-count registers, and finally unmasks it. PCDAC.SETUPDMA presumes that BUSDMA.INIT has been called.

PAGE is the high 4 bits and address the low 16 bits of the 20-bit base address of the dma buffer in the external-bus memory. NPOINTS is the number of 16-bit data points to be transferred (minimum

1, maximum = default = 32768). WRITETOMEM? specifies the direction of the transfer: NIL for PCDAC.STARTWRITED/A, non-NIL for PCDAC.STARTREADA/D. AUTOINIT? is NIL if the dma controller is to stop honoring dma requests when NPOINTS data points have been transferred; it is non-NIL if the dma buffer is to be used as a circular buffer, with address and transfer-count registers automatically reinitialized every NPOINTS data points; in this latter case the dma controller does not of itself stop honoring dma requests. See the documentation for the BUSMASTER library package for more details.

PCDAC.IOADDRESS

A global variable which determines the i/o addresses the PCDAC functions will use for the DT2801 series board. Initially 2EC(hex). Must be SETQed equal to the data register i/o address set by jumpers on the DT2801 series board, if different.

PCDAC.DMACHANNEL

A global variable which determines the dma channel number that PCDAC.SETUPDMA will use. Initially 1. Must be SETQed equal to that set by jumpers on the DT2801 series board, if different.

A/D Functions

The argument GAINCODE in the following functions encodes the analog input gain:

GAINCODE	gain on DT2801xxx board	gain on DT2805xxx board
0	1	1
1	2	10
2	4	100
3	8	500

(PCDAC.READA/DIMMEDIATE GAINCODE CHANNEL EXTTRIGGER?)

Performs a single A/D conversion on the channel at the analog input gain encoded by GAINCODE (described above), returning the result as a positive integer. This function encapsulates the DT2801 series boards' command sequence Read A/D Immediate.

(PCDAC.SETA/DPARAMETERS GAINCODE STARTCHANNEL ENDCHANNEL NPOINTS)

Determines several parameters for subsequent PCDAC.STARTREADA/D function calls. GAINCODE encodes the analog input gain settting, as described above. The channels from STARTCHANNEL to ENDCHANNEL inclusive are scanned cyclically, one channel being sampled at each clock. (ENDCHANNEL defaults to STARTCHANNEL.) NPOINTS is significant only if the PCDAC.STARTREADA/D function is called with CONTINUOUS? = NIL; then it is the number of A/D conversions (one channel each) to take place during that Read A/D command (minimum = 3, maximum = default = 65535). This function encapsulates the DT2801 series boards' command sequence Set A/D Parameters.

(PCDAC.STARTREADA/D DMA? CONTINUOUS? EXTCLOCK? EXTTRIGGER?)

Initiates a scan of clocked A/D conversions on the channels and at the input gain specified in the last PCDAC.SETA/DPARAMETERS function call. If DMA? is specified non-NIL then the 16-bit data values will be deposited in external-bus memory by direct memory access according to the last PCDAC.SETUPDMA's NPOINTS; otherwise the data must be input a point at a time with PCDAC.READA/DDATUM. If CONTINUOUS? is specified non-NIL then A/D conversions will take place until a Stop command is issued to the DT2801 series board; otherwise the number of conversions that takes place is governed by the last PCDAC.SETA/DPARAMETERS call. If EXTCLOCK? is specified non-NIL then conversions are clocked by an external signal supplied to the board; otherwise the board's internal clock is used, at the clock rate specified in the last PCDAC.SETCLOCK call.

This function encapsulates the DT2801 series boards' command sequence Read A/D With DMA and, with PCDAC.READA/DDATUM, the command sequence Read A/D.

Note that, when PCDAC.STARTREADA/D is called with DMA? specified non-NIL, both the Read A/D command and the dma operation can have had word counts specified, or either or both can have been specified to have no effective word count. The hardware treats the two counters completely separately. The simplest thing is to specify both word counts the same and to call PCDAC.SETUPDMA before each PCDAC.STARTREADA/D call. However exotic results, some of them quite useful, can be got with other combinations. Again, see the documentation for the BUSMASTER library package.

(PCDAC.READA/DDATUM)

Inputs individual data points during a Read A/D command started by PCDAC.STARTREADA/D with DMA? = NIL. The datum is returned as a positive integer. Coordination with

PCDAC.SETA/DPARAMETERS' NPOINTS and with the clock may be required; the functions PCDAC.READYFORCOMMAND? and PCDAC.READYFORREAD? may be of use in this. Note that the "Read A/D" command sequence described in the DT2801 series User Manual actually corresponds to PCDAC.STARTREADA/D plus one PCDAC.READA/DDATUM.

D/A Functions

(PCDAC.WRITED/AIMMEDIATE DACSELECT DATUM SECONDDATUM EXTTRIGGER?)

Performs a single D/A conversion on DAC channels 0, 1, or both simultaneously (as DACSELECT is 0, 1, or 2). If only one channel is selected, then the less significant bits of the integer DATUM goes to that channel, and SECONDDATUM is ignored if present; if both channels are selected, then DATUM goes to channel 0 and SECONDDATUM to channel 1. This function encapsulates the DT2801 series boards' command sequence Write D/A Immediate.

(PCDAC.SETD/APARAMETERS DACSELECT NPOINTS)

Determines two parameters for subsequent PCDAC.STARTWRITED/A calls. DACSELECT determines whether output is to channel 0 only, channel 1 only, or both channels simultaneously (as DACSELECT is 0, 1, or 2). If DACSELECT is 2, then data are output in pairs, the first datum of each pair going to channel 0 and the second to channel 1. NPOINTS is significant only if PCDAC.STARTWRITED/A is called with CONTINUOUS? = NIL; then it is the number of D/A conversions (one or both channels) to take place during that Write D/A command (minimum = 3, maximum = default = 65535). This function encapsulates the DT2801 series boards' command sequence Set D/A Parameters.

(PCDAC.STARTWRITED/A DMA? CONTINUOUS? EXTCLOCK? EXTTRIGGER?)

Initiates a scan of clocked D/A conversions on the DAC or DACs specified in the last PCDAC.SETD/APARAMETERS call. If DMA? is specified non-NIL then the 16-bit data values will be read from external-bus memory by direct memory access according to the last PCDAC.SETUPDMA call; otherwise the data must be output a point at a time with PCDAC.WRITED/ADATUM. If CONTINUOUS? is specified non-NIL then D/A conversions will take place until a Stop command is issued to the DT2801 series board; otherwise the number of conversions that takes place is governed by the last PCDAC.SETD/APARAMETERS' NPOINTS. If EXTCLOCK? is specified non-NIL then conversions are clocked by an external signal supplied to the board; otherwise the board's internal clock is used, at the clock rate specified in the last PCDAC.SETCLOCK call.

This function encapsulates the DT2801 series boards' command sequence Write D/A With DMA and, with one or two PCDAC.WRITED/ADATUM calls, the command sequence Write D/A.

Note that, when PCDAC.STARTWRITED/A is called with DMA? specified non-NIL, both the Write D/A command and the dma operation can have had word counts specified, or either or both can have been specified to have no effective word count. The hardware treats the two counters completely separately. The simplest thing is to specify both word counts the same and to call PCDAC.SETUPDMA before each PCDAC.STARTWRITED/A call. However exotic results, some of them quite useful, can be got with other combinations. Again, see the documentation for the BUSMASTER library package.

(PCDAC.WRITED/ADATUM DATUM)

Outputs individual data points during a Write D/A command started by PCDAC.STARTWRITED/A with DMA? = NIL. The less significant bits of the integer DATUM are output. If both DACs are being written to, then pairs of data points must be output, with the first point of each pair going to DAC 0. Coordination with PCDAC.SETD/APARAMETERS' NPOINTS and with the clock may be required; the functions PCDAC.READYFORCOMMAND? and PCDAC.READYFORWRITE? may be of use in this. Note that the "Write D/A" command sequence described in the DT2801 series User Manual actually corresponds to PCDAC.STARTWRITED/A plus one or two PCDAC.WRITED/ADATUM calls.

Digital I/O Functions

(PCDAC.SETDIGITALINPUT PORTSELECT EXTTRIGGER?)

Configures digital i/o ports 0, 1, or both (as PORTSELECT is 0, 1, or 2) to accept digital inputs. This function encapsulates the DT2801 series boards' command sequence Set Digital Port For Input.

(PCDAC.SETDIGITALOUTPUT PORTSELECT EXTTRIGGER?)

Configures digital i/o ports 0, 1, or both (as PORTSELECT is 0, 1, or 2) to provide digital outputs. This function encapsulates the DT2801 series boards' command sequence Set Digital Port For Output.

(PCDAC.READDIGITALIMMEDIATE PORTSELECT EXTTRIGGER?)

Reads digital input from digital i/o ports 0, 1, or both. If PORTSELECT is 0 or 1 then one byte of data is read (from port 0 or 1) and returned as a small positive integer. If PORTSELECT is 2, then

two bytes of data are read simultaneously (one from each port) and returned as a 16-bit positive integer, with the port 1 data as the more significant byte. The ports selected must of course currently be set for input. This function encapsulates the DT2801 series boards' command sequence Read Digital Input Immediate.

(PCDAC.WRITEDIGITALIMMEDIATE PORTSELECT DATUM EXTTRIGGER?)

Writes digital output to digital i/o ports 0, 1, or both. If PORTSELECT is 0 or 1 then one byte of data is written (to port 0 or 1), namely the least significant 8 bits of the integer DATUM. If PORTSELECT is 2, then two bytes of data are written simultaneously: the least significant 8 bits of the integer DATUM are written to port 0 and the next less significant 8 bits are written to port 1. The ports selected must of course currently be set for output. This function encapsulates the DT2801 series boards' command sequence Write Digital Input Immediate.

PIPES

By: Mike Bird

INTRODUCTION

This package incorporates several components:

* A mechanism consisting of a ring-buffer connecting a pair of streams - similar to UNIX pipes.

* Some pipe-related functions, such as a TEE filter and a special form to create a pipeline - a number of processes whose primary input/output streams are connected together.

* A function called OPENRESETSTREAM which encapsulates RESETSAVE activity commonly associated with an OPENFILE or OPENSTREAM.

[Primary input and output streams are those accessed by specifying NIL to BIN and BOUT].

USER FUNCTIONS

(OPENPIPE SIZE)

Returns a dotted-pair of streams - the CAR of which can read a ring-buffer written via the CDR. SIZE must be in the range 1 to 32767 inclusive and defaults to 512. Both streams ignore DELETEFILE, GETFILEINFO, GETFILENAME and SETFILEINFO - returning NIL.

The read-stream can BACKFILEPTR, BIN, CLOSEFILE, EOFP and PEEKBIN. This is sufficient to support READ et.c. EOFP always waits until there is a byte in the buffer or the write-stream is closed. If the write-stream is closed BIN takes end of file action when the buffer becomes empty.

The write-stream can BOUT, CLOSEFILE and EOFP. This is sufficient to support PRINT et.c. EOFP always returns T. The write-stream is closed if the read-stream is closed, but not vice versa.

(PIPELINE FORMS)

Creates a new process for all forms except the last - which is evaluated in the current process. The first and last forms are evaluated with their primary input and output streams (respectively) taken from the current environment. The primary streams of the others are linked together by pipes.

Provision is made to close pipes on process termination, thus propagating EOF in the conventional manner.

In other respects PIPELINE behaves like PROGN.

*** WARNING ***

Interlisp-D has a tendency to drop into Raid sometimes when one process attempts to use another's primary streams. Therefore it is presently advisable to avoid having the first process in the pipeline read its primary input.

(TEE.FILTER TEEFILE APPEND?)

This function copies bytes from its primary input to its primary output and TEEFILE. If APPEND? is NIL, TEEFILE is opened in OUTPUT mode and deleted on error or abort. Otherwise TEEFILE is opened in APPEND mode and is not deleted.

(STREAM.SOURCE INPUT)

This function copies bytes from the specified input file to its primary output.

(STREAM.SINK OUTPUT APPEND?)

This function copies bytes from its primary input to the specified output. APPEND? is as for TEE.FILTER.

(FILTER.KEYBOARD INPUT OUTPUT STRING)

This function copies bytes from INPUT - which is typically T - to OUTPUT until STRING is encountered. STRING is not copied. STRING defaults to a single ↑Z character. [Because of line buffering it is necessary to type a newline sometime after the ↑Z in order for it to be seen].

[This function is needed because (EOFP T) is often true - so COPYBYTES won't work].

(ADD.FILTER FORM INPUT CLOSEINPUT OUTPUT CLOSEOUTPUT)

This function creates a process executing FORM with primary input INPUT and primary output OUTPUT. If CLOSEINPUT (resp. CLOSEOUTPUT) is non-NIL then a RESETSAVE is issued within the new process before evaluating FORM to ensure that INPUT (resp. OUTPUT) is closed when the process terminates.

(OPENRESETSTREAM FILE ACCESS RECOG BYTESIZE PARAMETERS DELETE-Y/N/NIL DELETE-ON-ANY-STATE)

FILE, ACCESS, RECOG, BYTESIZE and PARAMETERS are as for OPENSTREAM (which is similar to OPENFILE).

If FILE denotes an already open file/stream then, unless RECOG is NEW (or RECOG = NIL and ACCESS = OUTPUT), the stream associated with that file is checked for ACCESS compatibility and returned.

Otherwise a new stream is opened. A RESETSAVE is issued to ensure that the stream will eventually be closed and, if RESETSTATE is non-NIL, optionally deleted.

The file will be deleted iff:

```
(AND
  "It was not previously open"
  (OR
    "RESETSTATE is non-NIL"
    "DELETE-ON-ANY-STATE is non-NIL")
  (OR
    "DELETE-Y/N/NIL is Y"
    (AND
      "DELETE-Y/N/NIL is NIL"
      (OR
        "RECOG is NEW"
        (AND
          "RECOG is NIL"
          "ACCESS is OUTPUT")))))
```

EXAMPLES

```
(PIPELINE
  (STREAM.SOURCE 'A)
  (TEE.FILTER 'B)
  (STREAM.SINK 'C))
```

Copies file A to files B and C, although not very efficiently.

```
(FILTER.KEYBOARD NIL 'MYFILE)
```

Copies key input to MYFILE.

```
(APPLY
  'PIPELINE
  (APPEND
    '((PROGN
      (TTY.PROCESS (THIS.PROCESS))
      (FILTER.KEYBOARD T)
      (TTY.PROCESS T)))
    (for I from 1 to 20 collect '(COPYBYTES))))
```

Tests pipes and processes.

PLAY

By: Kelly Roach (Roach.pa @ Xerox.ARPA)

INTRODUCTION. The PLAY package offers Interlisp-D users a disciplined way to play simple musical melodies on Xerox 1108 machines. Typing (PLAY.DEMO) will demo the PLAY package to the user.

The main entry points to this package are functions

(PLAY.NOTES NOTES) and

(PLAY.MELODY MELODY).

Both these functions take a musical representation, a series of NOTEs or a MELODY, into a TUNE playable by Interlisp-D PLAYTUNE function.

To get some feel for this, a sample call to PLAY.NOTES might be

```
(PLAY.NOTES '(b b >c >d >d >c b a g g a b b a+ a b b >c >d >d
    >c b a g g a b a g+ g a a b g a B/ >c/ b g a
    B/ >c/ b a g a dx b b >c >d >c b a g g a b a
    g+ g))
```

and the PLAY package global DEMO.MELODY is the sort of argument taken by PLAY.MELODY.

Function (PLAY.KEYBOARD) is a way to catch up on your piano practice. We now proceed to the details. We first discuss the particulars of NOTEs and MELODYs, then describe the PLAY package functions which can be used with these objects.

NOTES

A NOTE is a litatom whose pname characters are interpreted as follows:

(1) KEY. "A", "B", "C", "D", "E", "F", "a", "b", "c", "d", "e", or "f" specifies the note key. If the note key is in upper case, the note is held for its full duration with no break (a tied or slurred note). Most notes are followed by a break and so are in lower case. Two additional keys, "R" and "r" play silently and are used as rests.

(2) OCTAVE. Each ">" raises the note an octave. Each "<" lowers the note an octave. Thus, Middle C, High C, and Low C are represented as C, >C, and <C. Each octave begins with key C and ends with key B (those are the rules!). A schematic of the great staff:

```
*---------- >F
*          >E
*---------- >D
*          >C
*---------- B
*          A
*---------- G
*          F
*---------- E
*          D
*          C
*          <B
*---------- <A
*          <G
*---------- <F
*          <E
*---------- <D
*  .       <C
*---------- <<B
*          <<A
*---------- <<G
```

(3) ACCIDENTALS. Each "#" (sharp) raises the note a semitone. Each "@" (flat) lowers the note a semitone. In MELODYs, "n" (natural) may also be used, (described further below). Accidentals only affect the note to which they are attached (i.e., they do not spread to other notes in a measure, etc., as they do in normal sheet music. You must explicitly put them where they are needed. MELODY KEY described below can be used to avoid most of the pain).

(4) DURATION. A note starts out as a quarter note. Each "/" halves the notes duration. Each "x" doubles the note duration. Thus: "xx" = full, "x" = half, "" = quarter, "/" = eighth, "//" = sixteenth, etc.

(5) ARTIFICIAL DIVISION. Digits "2", "3", "4", "5", "6", "7", "8", and "9" are used for artificial division: duplets, triplets, quadruplets, etc. When digit N occurs, the duration of the note is multiplied by the factor

```
(FQUOTIENT (FDIFFERENCE N 1.0) N)
```

Thus, the 2nd through 4th notes of the notes (C D3 E3 F3 G) can be viewed as a triplet.

(6) DOTS. One "+" will extend the length of the note by 50%. Two "+"s will extend the length of the note by 75%. And in general, N "+"s will extend the length of the note by the factor

```
(FQUOTIENT (FTIMES (FPLUS 1.0 (EXPT 2.0 (ADD1 N))))
           (EXPT 2.0 N))
```

You probably will never use more than two "+"s.

(7) BREAK. Each "↑" causes the duration of the note break to double (a staccato note). Each "←" causes the duration of the note break to halve.

MELODY

Knowing the details of NOTEs is enough to allow you to use the PLAY package relatively primitive PLAY.NOTES function effectively. More serious composition, such as playing melodies found in sheet music, is made facile through the use of MELODYs and function

PLAY.MELODY.

A melody is an instance of the record type

```
(TYPERECORD MELODY (TEMPO KEY METER BEAT PASSAGES))
```

Field PASSAGES of a MELODY is a list of PASSAGEs which are instances of the record type

```
(RECORD PASSAGE (REPEATS MEASURES))
```

The fields of these two record declarations are explained as follows:

(1) TEMPO of a MELODY is the speed with which the melody is played. TEMPO can be any of the litatoms ALLEGRO, FAST, MODERATO, MODERATE, ADAGIO, or SLOW, or MELODY can be a positive integer specifying the number of beats per minute for the melody. (ALLEGRO = FAST = 120, MODERATO = MODERATE = 90, and ADAFIO = SLOW = 60 beats per minute). The TEMPO for most melodies is MODERATE.

(2) KEY of a MELODY is the melody key signature. KEY can be any of the litatoms CMAJOR, GMAJOR, DMAJOR, AMAJOR, EMAJOR, BMAJOR, F#MAJOR, C#MAJOR, FMAJOR, B@MAJOR, E@MAJOR, A@MAJOR, D@MAJOR, G@MAJOR, or C@MAJOR. Or KEY can be a list whose CAR is either of # or @ and whose CDR elements are drawn from the list (C D E F G A B). The KEY of a MELODY determines which notes in the melody are automatically played sharp or flat. Accidentals occurring on a note cancel any effect KEY would have had on that note. Accidental "n" (natural) is used to just cancel the KEY effect.

(3) METER of a MELODY is the number of beats per measure for the melody, the numerator of the time signature you will find in sheet music (e.g., 3 in 3/4 time). METER isn't used much for anything right now by the PLAY package, but could be used later on as a check that the duration for each of your measures is correct. Usually METER is 4.

(4) BEAT of a MELODY is the kind of note being used as the beat for the melody, the denominator of a time signature. A BEAT = 4, 8, or 16 corresponds to quarter, eighth, or sixteenth notes. Usually BEAT is 4.

(5) PASSAGES of a MELODY is a list of PASSAGEs. Each passage in a melody contains fields REPEATS and MEASURES. If MAXREPEAT is the maximum repeat number occurring in the REPEATS fields of a melody passages, then MAXREPEAT passes will be made over the melody.

(6) REPEATS of a PASSAGE is a list of positive integers indicating on which passes through the melody this passage will be played. If positive integer I occurs in REPEATS, then the passage will be played on pass I.

(7) MEASURES of a PASSAGE is a list of measures. Each measure is a list of notes. In sheet music, measures are delimited by vertical lines. If you organize your notes into the same measures you find in your sheet music, you'll probably have an easier time at trying to edit your melody later on when you need to compare the melody against the sheet music.

PLAY FUNCTIONS

The patient reader is now in a position to use the functions the PLAY package has to offer. The following functions are made available to the user:

(1) (PLAY.NOTES NOTES) converts a list of NOTEs into a TUNE which can be played by PLAYTUNE.

(2) (PLAY.MELODY MELODY) converts a MELODY into a TUNE which can be played by PLAYTUNE.

(3) (PLAY.KEYBOARD) converts the user's keyboard into a primitive piano. The row of keys "ASDFGHJKL;'←" are played as <A, <B, C, D, E, F, G, A, B, >C, >D, and >E. Sharp keys can be found on the row above. Hitting the space bar causes PLAY.KEYBOARD to return the TUNE you played, which can be played by PLAYTUNE.

(4) (PLAY.TRANSCRIBE TUNE) can transcribe a TUNE produced by PLAY.KEYBOARD. (Duration information is just thrown away since there does not seem to be a satisfactory way of getting it right).

(5) (PLAY.ADJUST.TEMPO TUNE FACTOR) can be used to adjust the speed of a TUNE. TUNE speed will be uniformly increased by FACTOR.

(6) (PLAY.ADJUST.PITCH TUNE SEMITONES) can be used to adjust the pitch of a TUNE. TUNE pitch will be uniformly increased SEMITONES semitones.

(7) (PLAY.MESA STRING) converts a Mesa PLAY package string into a list of NOTEs that can be played by PLAY.NOTES.

(8) (PLAY.DEMO) demos the PLAY package.

PQUOTE

This package causes (QUOTE expression) to print out as 'expression, i.e., with single quote, both to file and to terminal. It also moves the T readmacro to FILERDTBL so that such expressions will read in as (QUOTE expression).

PRINTOUT

By: Jeffrey Shulman (Shulman @ Rutgers.ARPA)

These new functions make the following changes in the current behavior of PRINTOUT.

- PRINTPARA uses pixel positioning when printing to displaystreams.
- Bitmaps are incorporated as a printout form. This new form is .BM<before><descent><bitmap> which is implemented by the function PRINTBM. <before> is the amount to scroll up the displaystream before bitmap is displayed. NIL means don't scroll, T means scroll the amount the bitmap extends above the top of the current line, a number means scroll that amount above the top of the current line. <descent> is how far below the current line to BITBLT the bitmap. The window is scrolled up if there is not enough room. The following line printed will be printed so it doesn't overlap the bitmap. NIL means don't descend, a number means descend that amount. <bitmap> is the bitmap to print.

This displaystream is left at the same Y position it was before the BITBLT and advanced the width of the bitmap. If the bitmap is too wide to fit, a new line is generated.

PROFILETOOL

By: Martin Yonke (Yonke @ USC-ECL.ARPA)

The Profile Tool is used to create a graphical user interface to attributes of any "package" written in Interlisp. It accomplishes this by creating a specialized InspectorWindow for enumerating and changing variables, etc., of a package as described to the ProfileTool. (It assumes the user is familiar with the use of InspectorWindows.)

Use

A Profile Tool is created by calling the function PROFILETOOL with ProfileKeys and title for the InspectorWindow. ProfileKeys is a list of instances of the record PROFILEKEY. A PROFILEKEY record has the following fields: DESCRIPTOR, FETCHFN, REPLACEFN, HELPSTRING, and EXTRADATA.

DESCRIPTOR is usually an atom (although it can be a string) which gets printed as the "property name" in the InspectorWindow.

FETCHFN is a function to be evoked to determine the "value" of the DESCRIPTOR. If FETCHFN is a litatom, then it will be applied (APPLY*) to the DESCRIPTOR and PROFILEKEYS. If FETCHFN is NIL, then (GETTOPVAL DESCRIPTOR) will be used.

REPLACEFN is a function to be evoked to "set" the value of the DESCRIPTOR. If REPLACEFN is a litatom, then it will be applied to DESCRIPTOR and PROFILEKEYS (unless it is the atom DON'T, in which case there is no REPLACEFN). It should determine a value, usually by asking the user, and return the new value. If REPLACEFN is a list of litatoms, then that list will be taken as the list of possible values for the DESCRIPTOR, a pop-up menu will be created of those items, and a SETTOVAL will be evoked on the DESCRIPTOR and the value returned from MENU. If REPLACEFN is NIL, then the behavior will be as if REPLACEFN was (T NIL) - i.e., the only possible values of DESCRIPTOR are T or NIL. Whenever REPLACEFN is used, the new value will be redisplayed in the Profile Tool window, expanding its width if necessary.

EXTRADATA is a place holder for any "package information" associated with the DESCRIPTOR. Since PROFILEKEYS are passed to FETCHFN and REPLACEFN, a user of ProfileTool can use this as he pleases (e.g., to cache information).

Some Examples

If a package has a global variable that should only have the values T or NIL, then the key in PROFILEKEYS can be just the value of (create PROFILEKEY DESCRIPTOR ← 'var).

If a package has a globalvariable that should only have the values FOO, FUM, or FIE, then the key should be the value of

```
(create PROFILEKEY    DESCRIPTOR    ← 'var
                      REPLACEFN     ← '(FOO FUM FIE).
```

If a package wants to display a read-only item, e.g., the user's name, then the key should be the value of

```
(create PROFILEKEY    DESCRIPTOR    ← 'UserName
                      FETCHFN       ← (FUNCTION GETUSERNAME)
                      REPLACEFN     ← 'DON'T)
```

where GETUSERNAME is defined to be LAMBDA (D K) (USERNAME NIL T).

Of course, the FETCHFN and REPLACEFN can be as complex as necessary. Profile Tool has been designed to give maximum flexibility, but simplifying the most typical cases - i.e., global variables with a fixed set of possible values.

PROMPTREMINDERS

By: Jon L. White

Using the Interlisp facility of PROMPTCHARFORMS, one may be periodically "reminded" of important things by a message which is aggressively "winked" and "flashed" in the PROMPTWINDOW (or on primary output for systems without bitmap display, such as Interlisp-10 and Interlisp/VAX). It will desist the wink/flash "hassling" only after it has been acknowledged by user response, or after a pre-set interval of "hassling" time has elapsed.

REMINDERS is a filepkg type, so that they may be easily saved on files, and so that the general typed-definition facilities may be used. On any file which uses the REMINDERS filepkgcom, it is advisable to precede this command with a (FILES (SYSLOAD COMPILED FROM LISPUSERS) PROMPTREMINDERS) command, since this package is not in the initial Lisp loadup. When initially defining a reminder, it is preferable for the user to call SETREMINDER rather than PUTDEF; but HASDEF is the accepted way to ask if some name currently defines a "reminder", and DELDEF is the accepted way to cancel an existing "reminder".

In the first example below, the user wants to be reminded every 30 minutes that he ought to be using MAKEFILE to save his work; in the second example, he merely wants to be told once, at precisely 4:00PM to call home. Examples:

```
(SETREMINDER NIL (ITIMES 30 60) "Have you MADEFILE recently?")

(SETREMINDER 'WOOF NIL "Don't forget to inform wife of dinner plans."
                "8-Jan-83 4:00PM")
```

Functions

(SETREMINDER NAME PERIOD MESSAGE INITIALDELAY EXPIRATION)

This will create and install a "reminder" with the name NAME (NIL given for a name will be replaced by a gensym), which will be executed every PERIOD number of seconds by winking the string MESSAGE into the prompt window (if MESSAGE is null, then NAME is winked). "Winking" means alternately printing the message and clearing the window in which it was printed, at a rate designed to attract the eye's attention.

The first such execution will occur at PERIOD seconds after the call to SETREMINDER unless INITIALDELAY is non-NIL, in which case that time will be used; a fixp value for INITIALDELAY is interpreted as an offset from the time of the call to SETREMINDER, and a stringp value is interpreted as an absolute date/time string.

If PERIOD is null, then the reminder is to be executed precisely once. If EXPIRATION is non-null, then a fixp means that "expiration" number of seconds after the first execution, the timer will be deleted; a stringp means a precise date/time at which to delete the timer.

Optional 6th and 7th arguments -- called REMINDINGDURATION and WINKINGDURATION -- permit one to vary the amount of time spent in one cycle of the wink/flash loop, and the amount of time spent winking before initiating a "flash". The attention-attracting action will continue for REMINDINGDURATION seconds (default: 60), or until the user types something on the keyboard; care is taken not to consume the typed character *except* when it is a space (which is just tossed out). Type-ahead does not release the winking. In case the user has become too drowsy to notice the winking, then every WINKINGDURATION seconds (default: 10) during the "reminding", the whole display videocolor will be wagged back and forth a few times, which effects a most obnoxious stimulus (for non-bitmap systems, this just types a <bell>).

Returns the name (note above when NIL is supplied for the name).

(ACTIVEREMINDERNAMES) No arguments; self-explanatory.

(REMINDER.NEXTREMINDDATE NAME) Returns the time (in GDATE format) at which the next reminding from the named reminder will occur; NIL if NAME isn't a REMINDERS. (REMINDER.NEXTREMINDDATE NAME Date/Time.string) Sets the time at which the reminder is next to be executed.

(REMINDER.EXPIRATIONDATE NAME) Returns the time (in GDATE format) at which the reminder will be automatically deleted. (REMINDER.EXPIRATIONDATE NAME Date/Time.string) Sets the expiration time.

(INSPECTREMINDER NAME) In Interlisp-D, this will call INSPECT on the definition of the named reminder; in other systems, it merely calls SHOWDEF.

QIX

By: Jeff Shrager

INTRODUCTION

QIX is a small graphic demo modelled after the videogame of the same name. To start a QIX, call (QIX.GROW <window> <dontdismiss>). This starts a bouncer in the indicated window (if no window is specified, the user is prompted for a window). If <dontdismiss> is non-NIL, QIX.GROW will not dismiss between drawing each line. The QIX will follow the edges of the window, in case it is reshaped.

(QIX.PLAY <n>) will open a screen-sized window and run n QIX's in it. This actually starts up n processes so they have to be killed via the PSW...which can be annoying. Again, reshaping the large window will cause the QIX's to follow its edges appropriately.

A DLion can handle about 2 or 3 QIX's before they start slowing each other down.

RECORDER

By: Jeffrey S. Shulman (Shulman @ Rutgers.ARPA)

Introduction

RECORDER is a package that allows you to record and playback mouse and keyboard events. These events are recorded as you perform them in more or less [N.B. Due to the way playback occurs interrupts will not be seen in real time. For example, if you were pretty-printing a long function to the typescript window and ↑E'ed it in the middle, the printing would (during recording) stop immediately. However, during playback, the ↑E would not be seen until the pretty-printing was finished.] real time. During playback these recorded events are re-performed as if you were there doing it again.

RECORDER performs these feats of magic by redefining the low level mouse and keyboard handlers. Specifically RECORDER provides its own versions of GETMOUSESTATE and \GETSYSBUF (as well as a slightly modified version of \KEYHANDLER1).

Use

To use you first load RECORDER.DCOM. After this file is loaded it will BKSYSBUF its initialization function as well as a HARDRESET (this is necessary in order for the modified \KEYHANDLER1 to take affect).

Starting a recording

To start a recording session you use the function RECORD.START. This function returns immediately with the array the recording will be placed in. This array is a SMALLP array of \RASIZEelements.

At this point the recording has not actually started. You should now position the mouse where you want it to be when the recording starts. To actually start the recording you press the CONTROL and LEFT SHIFT key simultaneously (this fact also appears in the PROMPTWINDOW after you call RECORD.START as a reminder.)

When these keys are pressed the recording starts. The message Recording.... will appear in the PROMPTWINDOW. Recording proceeds until either you press CTRL-LSHIFT again or the array fills up. When either of these events occur the message stopped. will appear in the PROMPTWINDOW.

A sync event (described below) is recorded automatically when you start and when you stop a recording.

Thus, an example of the sequence of events would be:

(SETQ A (RECORD.START))
position the mouse to the starting position
Press CTRL-LSHIFT
perform the desired actions
Press CTRL-LSHIFT

This array may be saved on a file using the UGLYVARS *File Package* command. The function SQUEEZE.RECORDING, described below, can be used to eliminate much of the dead space.

Playing back a previous recording

To playback a previous recording you may use the function REPLAY whose single argument is the array returned by a previous call to RECORD.START. REPLAY will set in motion what is needed to playback an old recording and will return after the playback is finished.

During playback the mouse and keyboard will NOT respond to anything you do. The only exceptions to this are:

* Pressing CTRL-LSHIFT will stop playback and return control to you.

* Pressing CTRL-LSHIFT-DELETE (the emergency interrupt) will stop playback and reset the system.

You will again receive control of the keyboard and mouse when the playback is finished.

If you typed what was in the previous "How to record" example you would play it back via:

(REPLAY A)

Synchronization

During playback you may wish to perform some "outside" event. For example while the mouse was moving around or when choosing various menu items you might want to print some accompanying text. An easy method of synchronizing just these kinds of things is provided so that things "happen" at the appropriate moment.

During playback this synchronization is accomplished by the function RP.SYNC. If while playback is happening a sync event occurs (more about how to put one in below) the recording will essentially suspend itself until a call to RP.SYNC has been performed. If a call to RP.SYNC occurs before a *sync event* it will not return from it until this *sync event* happens.

A *sync event* is automatically recorded at the start and at the end of each recording.

The function REPLAY is really a call to PLAYBACK.START followed by two calls to RP.SYNC (one for the start of the recording and one for the ending.)

The function PLAYBACK.START (whose single argument in the recording array) is what *really* starts playback of a previous recording. This function queues up the recording for playback and returns *immediately*. You should use this function and provide your own RP.SYNC's if you want to "do" something during a playback.

This should be come clear by an example. Suppose you wanted to print "I will now move to menu X", move to the menu, print "Now I will select an item from this menu" and then select an item from this menu.

During recording you would move the mouse over to the menu, cause a *sync event*, and then select the menu item.

A function that would playback this back correctly would look like:

```
(LAMBDA (RECORDED-ARRAY)
(PLAYBACK.START RECORDED-ARRAY)
(printout T "I will now move to menu X" T)
(RP.SYNC)
(RP.SYNC)
(printout T "Now I will select an item from this menu" T)
(RP.SYNC))
```

What happens in this function is:

* The recording is queued up. Since the act of making the recording puts in a sync event at the start, the mouse does not move.

* "I will now move to menu X" is printed.

* The first (RP.SYNC) starts the playback. The mouse begins moving to the menu.

* The second (RP.SYNC) does not return until the second *sync event* happens. This is the one that was intentionally put in during recording.

* "Now I will select an item from this menu" is printed after the second *sync event*.

* The third (RP.SYNC) waits for the last *sync event* that was automatically recorded when the recording stopped to happen.

Sync events

Sync events are caused to occur during recording by pressing the *sync* key. This key's number (returned by \KEYNAMETONUMBER) is bound to the global variable \SYNC.KEY. The default is 91 which is the AGAIN key on the Dandelion's keyboard.

When this key is pressed during recording a *sync event* is recorded at that instant. Feedback is provided by printing a letter S in the PROMPTWINDOW each time this key is pressed.

N.B.: If you are not careful it is possible to have the wrong number of *sync events* to the number of RP.SYNC function calls. This could cause the mouse to freeze waiting for a RP.SYNC function call (which you cannot type in yourself since the keyboard is locked out during playback.) Should this happen it is possible to stop playback and regain control via the CTRL-LSHIFT abort.

Caveats

It should go without saying that it is important for the *same exact* set of circumstances to exist before any given playback as they did when the recording was made.

Little *gotcha's* will pop up in the unexpected places so it is important to "set up" before any playback.

An example of a *gotcha* are popup menus that use the MENUOFFSET field to determine where they will next appear. If this is different at playback time from what it was at record time the results will not be the same (N.B.: the global menus WindowMenu, IconWindowMenu and BackgroundMenu are examples menus that use the MENUOFFSET field. Since these are the most commonly used menus that are all set to NIL (so they will be recreated from their corresponding *Commands* list) by both RECORD.START and PLAYBACK.START.)

Miscellaneous stuff

During playback mouse key presses are indicated by the letter L, M and R shown along the bottom of the cursor bitmap (at the obvious positions.)

The function SQUEEZE.RECORDING is provided to eliminate *dead space* at the end of a recording array. Its single argument is an array previously returned by RECORD.START. Its value is a new array of minimum size.

REGION

By: C. D. Lane (Lane @ SUMEX-AIM)

REGION.DCOM facilitates having multiple complex cursor behaviors in a single window without having the CURSORMOVEDFNs, CURSORINFNs, CURSOROUTFNs, and BUTTONEVENTFNs of the behaviors know about each other. In its simplest form it can be used to implement active regions of varying shapes and sizes.

To use REGION, set the various window functions of the window to the REGION window functions and put a list of REGIONEVENT records on the REGIONEVENTLST property of the window. When the cursor moves over the window, REGION checks which region it is in, calls the CURSOROUTFN of the previous region and the CURSORINFN of the new region. This is similar to having several windows within a window.

The REGION window functions are:

```
(WINDOWPROP WINDOW 'CURSORINFN (FUNCTION REGIONINFN))
(WINDOWPROP WINDOW 'CURSOROUTFN (FUNCTION REGIONOUTFN))
(WINDOWPROP WINDOW 'REPAINTFN (FUNCTION REGIONREPAINTFN))
(WINDOWPROP WINDOW 'CURSORMOVEDFN (FUNCTION REGIONMOVEDFN))
(WINDOWPROP WINDOW 'BUTTONEVENTFN (FUNCTION REGIONEVENTFN))
```

On the REGIONEVENTLST property of the window put a list of REGIONEVENT records which have the fields:

EVENTREGION A REGION record which is the region of the window over which the following functions will be invoked.

REGIONBUTTONFN The BUTTONEVENTFN(WINDOW POSITION) for the region.

REGIONMOVEDFN The CURSORMOVEDFN(WINDOW POSITION) for the region.

REGIONINFN The CURSORINFN(WINDOW REGION) for the region.

REGIONOUTFN The CURSOROUTFN(WINDOW REGION) for the region.

REGIONREPAINTFN The REPAINTFN(WINDOW REGION) for the region.

All of the fields in the REGIONEVENT record are optional.

DISABLEFLG

The variable DISABLEFLG, if T, disables all of the region functions for all of the window using the REGION package. Alternatively, setting DISABLEFLG to a window, or list of windows, disables all the windows using the REGION package except for those windows on DISABLEFLG. This allows turning off parts of the screen that should only be active at certain times.

SETF

By: Kelly Roach (Roach.pa @ Xerox.ARPA)

SETF (SET Field) modifies the Record Package slightly to get the user the MACLISP SETF-style record accessing. Less verbose than CLISP fetch and replace, but without the drawbacks of packing record accesses into atoms as with CLISP infixes colon (:) and period (.).

(1) RECORD ACCESS. For any record name "r" and field name "f", SETF makes the following translations:

(r.f d) becomes (fetch (r f) of d)

(SETF (r.f d) v) becomes (replace (r f) of d with v)

More precisely, these forms translate to what the fetch & replace forms translate to.

(2) ARRAY ACCESS. SETF can be used with ELT:

(SETF (ELT ARRAY N) V) becomes (SETA ARRAY N V)

(3) CAR & CDR ACCESS. SETF can be used with CAR & CDR:

(SETF (CAR X) Y) becomes (RPLACA X Y)

(SETF (CDR X) Y) becomes (RPLACD X Y)

(4) USER HOOK. The user can define his own SETF translations using property SETFDEF (This is how SETF for ELT, CAR, and CDR can be implemented:

(PUTPROP 'CADR 'SETFDEF '(RPLACA (CDR DATUM) NEWVALUE))

(PUTPROP 'ASSOC 'SETFDEF 'FOOFN)

then SETF will make the translations

(SETF (CADR X) Y) to (RPLACA (CDR X) Y)

(SETF (ASSOC KEY ALST) V) to
what (FOOFN '(ASSOC KEY ALST)) evaluates to.

In general, if the user does

(PUTPROP a 'SETFDEF setfdef)

then SETF makes the translation

(SETF (a d) n) to setdef with d & n substituted for DATUM & NEWVALUE if "setfdef" is a list and

(SETF (a ...) ...) becomes what (setfdef '(SETF (a ...) ...)) evaluates to if "setfdef" is a litatom.

SHOW

By: Beau Sheil and Ron Kaplan
(Sheil.pa @ Xerox.ARPA and Kaplan.pa @ Xerox.ARPA)

This is a trivial file that adds the following facilities:

It adds a LISPXMACRO SHOW (and lowercase show) that will pretty-print to the terminal the value of the preceding history event. It may be given event-specification arguments like ?? and REDO to indicate an event other than the last one whose value is to be printed.

The file also defines the function SHOW, which is used by the LISPXMACRO, and makes the function IT be equivalent to VALUEOF. Thus, IT as a variable is equivalent to (VALUEOF -1). (IT A B C) is equivalent to (VALUEOF A B C), but is easier to type.

The value of the SHOW macro and function is the item that it prints.

SIGNAL

By: Henry Thompson

The file SIGNAL.DCOM implements a first pass at the CEDAR/MESA signal mechanism. It allows signals to be raised, caught, examined, and resumed or exited. Catch phrases are introduced with the clisp-word **enable**; signals are raised with the function Signal.

(Signal type arg)

Raises a signal of type *type*. If the signal is resumed, will return with the argument to **sresume**, - see below. Otherwise will not return.

```
(enable
  s1 => a1 a2 a3 ...
     . . .
  sn => n1 n2 n3 ...
 form
  l1 -> la1 la2 ...
   . . .
  ln -> ln1 ln2 ...)
```

The double arrow lines above are called *catch phrases*, the single arrow lines are called *finish phrases*. Evaluates *form* so as to catch signals *s1, ... sn* if they are raised during its evaluation. If e.g. *s1* is raised, the forms a1 ... an
(the *catch phrase* for s1) will be evaluated in the context of the call to Signal which raised *s1*, with the addition of the fact that the variables type and arg will be locally bound to *type* and *arg*. For a catch phrase to be well formed, all control paths through it must end with one of the following four *quit forms*:

(exit)

Causes the stack to unwind back through the enclosing enable form, which is exited with value NIL.

(sresume form)

Returns from the call to Signal with the value of form as the value of that call.

(goto label)

Causes the stack to unwind back to the enclosing enable form, where the *finish phrase* for label is evaluated. The value of the last form in the phrase is the value of the enable. The variables $SignalType$, $SignalArg$, and $Exit$ will be bound to *type* and *arg* of the original call to Signal and *label* respectively, but otherwise the environment of the call to Signal is lost.

(reject)

Causes the signal handling process to act as if the catch phrase had not been there at all.

When a signal is raised the stack is scanned upwards for a catch phrase for that signal. If none is found (or if all those found are **rejected**) an Uncaught Signal break will occur. Otherwise the one of the three other options listed above will occur.

There is one signal name which receives special interpretation. A catch phrase with the name **any** will catch any signal not explicitly caught elsewhere in the enable.

There is a special label whose name is **unwind**, which cannot be gone to and has a special meaning. The finish clause for the **unwind** label will be evaluated as the stack unwinds upwards from a call to Signal to the enable clause which caught it (or to the UserExec if ERROR! is envoked).

At the moment the package does use ERROR! and NLSETQ itself to implement the stack unwinding. This means the package interacts correctly with RESETSAVE, but has the unfortunate consequence that interleaving on the stack of calls to ERRORSET in any of its forms and enable clauses has consequences which are confusing at best.

As there are many ways in which the semantics of signals and catchers are much cleaner than those of errors and errorsets, it is hoped that at some point a more sophisticated implementation of the signal mechanism undoing the resetsaves directly and redefining ERRORSET in terms of enable will be made available.

As an interim measure however, a mechanism for turning errors into signals has been provided. The function (MakeErrorsSignals) will arrange that all errors will be converted into a signal with name LispError and argument an instance of the following record:

(RECORD LispError (eMess eFn eType . ePos))

where eMess is the standard error message pair (for printing with ERRORMESS), eFn the name of the function which the old package would have broken, eType its type, and ePos a stack pointer to its frame. If you catch this signal, you should free the stack pointer. If you don't catch it (or catch and reject it), a break *will happen*, regardless of HELPDEPTH, ERRORSETs, etc. That is, BREAKCHECK is *not* called or paid attention to. This is the interim price of having the other benefits of the signal package, and in practice seems to work quite well. Things from the old package that *do* still work are ERRORTYPELST, user interrupts, and the various other funnies to do with hash arrays and EOFs which the error code handles.

To go back to the old way of life, use (MakeErrorsErrorsAgain).

There follows on the next page an toy example of the use of the package, which is in fact included in the file itself - enjoy!

```
(ST
 [LAMBDA NIL                                    (*.ht: "20-JAN-83 21:40")
  (enable
     s1 => (PRINT "s1 caught" T)
       (goto s1)
     s2 => (PRINT "s2 caught")
       (sresume 37)
     s3 => (PRINT "s3 caught")
       (reject)
     s4 => (PRINT "s4 caught")
       (exit)
     any => (printout T type " caught by any" T)
       (exit)
   (TestSignals)
     s1 -> (PRINT "s1 unwound")
     unwind -> (PRINT "unwinding"))])
```

```
(TestSignals
 [LAMBDA NIL                                    (* ht: "18-JAN-83 16:39")
  (printout T T
     (SELECTQ (PROGN (printout T T ">")
                     (READ))
          (1 (Signal 's1 1))
          (2 (Signal 's2 2))
          (3 (Signal 's3 3))
          (4 (Signal 's4 4))
          (5 (Signal 'foo 5))
          6)
     '←])
```

SNAPSCROLL

By: Jeff Shrager (Shrager @ CMU-CS-A.ARPA)

Loading SNAPSCROLL advises (SNAPW) so that snapshot windows are henceforth shapeable and scrollable. This enables the user to, for example, snap a long list off the screen, and then reshape it to a reasonable size and scan it at will.

NOTE:

This function advises SNAPW so software that relies on that FN may be affected.

A bitmap the size of the original snap window is constructed by snapscroll, so reshaping a window doesn't save any array space. In fact, you spend twice the space, for each snapwindow.

SPACEWINDOW

By: Michael Sannella (Sannella.pa @ Xerox.ARPA)

Loading SPACEWINDOW.DCOM puts a small "Space Allocation" window in the lower-right-hand corner of the screen. This window shows a bar-chart of the amounts of the four types of memory space that have been allocated (fixed data, variable data, atoms, pnames). This display will be updated every 60 seconds. It is a graphic version of the last part of the (STORAGE) printout.

The window is positioned at the value of SPACEWINDOWPOSITION, initially (675 . 0). You can set this variable before loading SPACEWINDOW if you like your window in a different place.

STARBG

BY: Gregg Foster and Richard Acuff (Acuff @ Rutgers.ARPA)

When STARBG is loaded, functions for painting a starfield onto the background are made available:

(STARBG) will create a new star field and make it the background.

(GENERATE.STARSCAPE) returns a new screen-sized bitmap which can be given to CHANGEBACKGROUND.

(TWINKLE) is similar to STARBG, except that the starfield is built dynamically while-you-watch.

STYLESHEET

By: Tayloe Stansbury

INTRODUCTION

Stylesheets are collections of menus. These collections pop up all at once in a group. This group does not disappear until all menus in it have been dealt with, and the user signals that he is done.

Stylesheets are intended to be used in situations wherein the computer wants an answer to several related questions all at once. One example is font selection. To select a font, the user needs to specify font family (Classic, Modern, etc.), font size (8 point, 10 point, etc.), and font style (bold, italic, etc.). Rather than prompt for each of these parameters in succession, one could use a stylesheet to prompt for it all at once.

When the stylesheet pops up, it will shade (preselect) default selections (if provided) in each of the menus. The user can either decide that the default selections are OK, or change them to suit his taste. (The default selection mechanism can be used to convey the current state of something the user is trying to change with the stylesheet: for example, the current looks of the text with which the user is dissatisfied.)

When the user is finished, he hits the DONE button and the stylesheet disappears, and the final selections are returned. There is also a RESET button. This is useful if the user has mucked up his selections and would like to reinstate the default selections. Finally, there is an ABORT button that if selected returns NIL from STYLESHEET and is intended to provide the user with a convenient way of aborting the selection. Note: This means that NIL can be returned from a call to STYLESHEET.

Menus in a stylesheet can be set up to accept exactly one selection (like a normal menu), less than two selections, or any number of selections. Menus that need not be filled in (i.e., can accept zero selections) have an attached CLEAR button, which can be used to remove selections made in that menu. Menus that can have more than one selection have an attached ALL button, which can be used to select all the items in the menu.

HOW TO MAKE A STYLESHEET

To create a stylesheet, call

(CREATE.STYLE *Prop1 Value1 Prop2 Value2 PropN ValueN*) [function]

CREATE.STYLE accepts an arbitrary number of property-value pairs. Properties currently recognised are

ITEMS: A list of menus. Most menu format parameters contained in menu records are honored by the stylesheet package. WHENSELECTEDFNs are, of course, ignored.

SELECTIONS: A list of menu items, each one corresponding to a menu in ITEMS. The specified selections will be shaded in the appropriate menu, and will be the default selections. If not specified or too short, it will be filled out with NILs (no selection).

NEED.NOT.FILL.IN: A list of T or NIL or MULTI, each one corresponding to a menu in ITEMS. T indicates that the corresponding menu need not be filled in and will be given a CLEAR button. MULTI indicates that the corresponding menu can have any number of selections and will be given both a CLEAR button and an ALL button. If the list is too short, it will be filled out with NILs. If a single T or NIL or MULTI is given instead of a list, it will be replaced by a list of Ts or NILs or MULTIs, respectively.

TITLE: The title that will be given to the stylesheet. If no title is specified, the stylesheet will not have a title bar.

ITEM.TITLES: A list of strings or atoms to serve as titles over the menus. Items without titles specified will not have titles.

ITEM.TITLE.FONT: A fontdescriptor or other font specification which determines the font item titles will be printed in. If NIL, titles will be printed in DEFAULTFONT.

POSITION: The screen position (of type POSITION) of the lower left-hand corner of the stylesheet. If position is not specified, the function STYLESHEET will prompt for the postion (using GETBOXPOSITION). STYLESHEET will modify positions as necessary to ensure that the entire stylesheet will be on the screen.

Stylesheets can be modified by calling

(STYLE.PROP *Stylesheet Prop Newvalue*) [function]

STYLE.PROP always returns the old value of the specified property of the specified stylesheet. If Newvalue is provided (even if NIL), it replaces the old value. If not provided, the old value remains. (Just like WINDOWPROP.)

To use the stylesheet thus created, call

(STYLESHEET *Stylesheet*) [function]

This returns a list of selections the user made from the stylesheet. (If a selection is returned as NIL, that indicates that no selection was made.)

One can determine in advance of displaying a stylesheet how big it will be. (This may help in determining a reasonable screen position for the stylesheet.) The relevant functions are

(STYLESHEET.IMAGEWIDTH *Stylesheet*) [function]

and

(STYLESHEET.IMAGEHEIGHT *Stylesheet*) [function]

They return the width and height, respectively, of the stylesheet in pixels.

AN EXAMPLE

The package is located in STYLESHEET and STYLESHEET.DCOM. To familiarize yourself with its workings, you might want to load it and try the following example:

```
(SETQ FONT.STYLE
 (CREATE.STYLE
  'TITLE "Please select a font:"
  'ITEM.TITLES '(Family Size Face)
  'ITEM.TITLE.FONT '(Modern 12)
  'ITEMS
   (LIST
    (CREATE MENU ITEMS ← '(Classic Modern Terminal))
    (CREATE MENU ITEMS ← '(8 9 10 11 12 14))
    (CREATE MENU ITEMS ← '(Regular Bold Italic BoldItalic)))
  'SELECTIONS '(Modern 11 Regular)
  'NEED.NOT.FILL.IN 'T]
(STYLESHEET FONT.STYLE]
```

SYSTAT

By: Kelly Roach (Roach.pa @ Xerox.ARPA)

This simple package redefines Interlisp function CONTROL-T so that you get a graphic display instead of the usual printed status line when you type character <CONTROL-T>.

The SYSTAT package puts up SYSTAT.WINDOW indicating USERNAME, TTY.PROCESS, STATE, STACK FRAME, %UTIL, %GC, %KEYBOARD, %SWAP, number of OPEN FILES, MAIN DATA SPACE used, and ARRAY SPACE used.

SYSTAT.WINDOW may also be usefully buttoned with your mouse. Buttoning with LEFT causes an immediate update of SYSTAT.WINDOW. A popup menu is available with MIDDLE allowing you to enter Quickly and Slowly update modes, immediate Update, Deactivate the SYSTAT.WINDOW, and Reactivate the SYSTAT.WINDOW. The ordinarily window menu is available with RIGHT. Closing the SYSTAT.WINDOW causes SYSTAT to deactivate until your next <CONTROL-T> which will reactivate SYSTAT, opening SYSTAT.WINDOW.