FORMULA Version 3.4 Reference Manual

David P. Anderson
Computer Science Division
University of California
Berkeley, CA 94720

Ron Kuivila
Music Department
Wesleyan University
Middletown, CT 06457

May 29, 1991

ABSTRACT

FORMULA is a Forth-based programming environment for computer music, and can be used for algorithmic composition, interactive systems, or programmed score interpretation. It provides a programming model in which separate lightweight processes are used to generate musical parameters such as pitch, rhythm, volume, articulation, and tempo. FORMULA runs on personal computers (Macintosh and Atari ST) and is typically used to control MIDI synthesizers. This manual describes the features of FORMULA and briefly discusses its implementation.
# TABLE OF CONTENTS

1. INTRODUCTION ............................................................... 1  
   1.1 Getting Started ....................................................... 1  

2. BASIC FEATURES .......................................................... 3  
   2.1 Playing Notes ........................................................ 3  
   2.2 Defining New Words ............................................... 3  
   2.3 Processes ............................................................ 4  
   2.4 Non-Standard Number Formats ................................... 6  

3. MORE FEATURES ............................................................ 7  
   3.1 Process Groups ...................................................... 7  
   3.2 Process Naming ...................................................... 7  
   3.3 Process Display and Control ..................................... 8  
   3.4 Per-Process Storage ............................................... 8  
   3.5 Specifying Rational Time Intervals .............................. 10  
   3.6 Vocabularies ......................................................... 11  

4. $\text{WORDS}$: A HIGH-LEVEL NOTE-PLAYING FACILITY ............ 12 
   4.1 Simple $\text{WORDS}$ .................................................. 12  
   4.2 Sustain Pedal ....................................................... 12  
   4.3 Synthesizer Output Paradigms .................................... 13  
   4.4 Pkans Used by the $\text{WORDS}$ ................................. 13  
   4.5 Compound $\text{WORDS}$ ............................................. 13  

5. AUXILIARY PROCESSES ................................................... 15 
   5.1 Introduction ......................................................... 15  
   5.2 Using Processes to Define Functions of Time .................. 17  
   5.3 Volume Control .................................................... 17  
   5.4 Tempo and Rubato Control using Time Deformations ............ 18  
   5.5 Articulation Control ............................................. 20  
   5.6 Timing Sequence Generators ..................................... 21  

6. PITCH SPECIFICATION ................................................... 23 
   6.1 Pitch Values and Pitch Indices .................................. 23  
   6.2 Symbolic Pitch Names ............................................ 23  
   6.3 Tuning Systems ..................................................... 23  
   6.4 Pitch Sets ......................................................... 24  

7. SYNTHESIZER OUTPUT ................................................... 26 
   7.1 Output Paradigms .................................................. 26  
   7.2 Synthesizer Configuration ...................................... 27  
   7.3 Direct MIDI Output ............................................. 29  
   7.4 The Dumb Synthesizer Manager ................................ 29  
   7.5 The Synthesizer Manager ....................................... 30  
   7.6 The MIDI Output Driver ......................................... 30  
   7.7 How to Write a Synthesizer Driver ............................ 31  

8. INPUT HANDLING .......................................................... 33
1. INTRODUCTION

FORMULA (Forth Music Language) is a programming environment for computer music. It runs on the Atari ST and is based on Forthmacs, a Forth system developed by Mitch Bradley. Distribution and copying policies for FORMULA and Forthmacs are given in Appendix A. A brief summary of Forth and Forthmacs is given in Appendix B.

FORMULA is intended for musicians who are also programmers. It emphasizes algorithms rather than static scores. FORMULA is not a sequencer or score editor, and currently has no graphical interface. However, using FORMULA you can write programs that:

- represent and perform "interpretations" of existing music;
- compose complex music using deterministic or random rules, and play it as they compose;
- interact with a human performer.

FORMULA programs produce sound by controlling an external synthesizer. Normally the connection to the synthesizer is made through MIDI (Musical Instrument Digital Interface; described in Appendix C). Programs can respond to input from various devices. On the Atari ST, these include the typewriter keyboard, the mouse, the joystick, and MIDI keyboards connected to the Atari's MIDI input port, as shown below.

![Diagram of FORMULA input and output devices](image)

1.1. Getting Started

The FORMULA Version 3.4 release is one double-sided 3.5" disk. The disk contains the source code for FORMULA (in the directory `formula.dir`), for synthesizer-specific configuration words (in `synths.dir`), and for various FORMULA programs (in `pieces.dir`). It also contains binaries for Forthmacs (`forth.tos`) and Emacs (`emacs.tos`), and a quick reference for Emacs (`emacs.ref`). Make a backup copy of the disk.

1.1.1. Compiling and Saving FORMULA

The FORMULA binary is not relocatable, so you must compile and save it on your own "working" disk (this disk must have at least 200K bytes free). To do this, first boot from the working disk. Then insert the FORMULA release disk, and run Forthmacs by double-clicking on `forth.tos`. When you get Forth's `OK` prompt, type
cd formula.dir
load load

FORMULA takes about two minutes to compile. When it’s done, type

cd ..\synths.dir
load config.gen

This loads a synthesizer configuration file (§7.2); config.gen is for a “generic” MIDI synthesizer with 8 voices on MIDI channel one. This can be customized to your own synthesizer configuration; see §7.

Now insert your working disk and type

cd ..
ls
"" formula.tos save-rel

This saves the FORMULA binary on disk as formula.tos. Henceforth you can load FORMULA directly from the working disk by double-clicking on formula.tos. You may also want to copy emacs.tos and the pieces.dir directory to the working disk.

1.1.2. Running FORMULA

Once FORMULA is loaded, you can start and stop it using

formula ( --- ; start FORMULA )
restore ( --- ; stop FORMULA )

Make sure your MIDI synthesizer is connected, and configure it to receive on MIDI channel one. You can now play a note by typing

$ 50

If this doesn’t work, something is wrong. Jiggle the cables, reboot, and try again.

The directory pieces.dir contains various FORMULA programs. Type

cd pieces.dir
load demo
rnd-notes

and you will hear a stream of random notes. When you get tired of this, type

kill-all

to stop it. The other files in the pieces.dir directory contain FORMULA programs in a variety of musical styles. Try compiling and running some of these (for fun, try running several at once). To hear a lengthy FORMULA sampler, type

load concert \ this will take a while
concert

1.1.3. Panic Buttons

If things go wrong and you lose control of the computer, press the undo key to abort FORMULA, and type formula again to continue. If this fails, press the Atari’s reset button and start over.

---

1 For obscure reasons, Emacs must not be loaded when you save FORMULA. Also, you can make an auto-booting version of FORMULA as follows: 1) create a directory named auto; 2) copy forth.tos into auto, renaming it auto.prg; 3) reboot from this disk; 4) load FORMULA and save it as auto.prg within auto.

2 If notes continue to sound indefinitely after kill-all, it is because one of the programs had a “sustain pedal” down. Type pedoff to release it.
2. BASIC FEATURES

2.1. Playing Notes

Commands can be sent from the Atari's MIDI output port using the following words:\(^3\):

- mku (velocity key-no channel --- ; key-up command)
- mkd (velocity key-no channel --- ; key-down command)
- mpp (pressure key-no channel --- ; polyphonic pressure)
- mcc (value control-no channel --- ; controller change)
- mpc (patch-no channel --- ; patch change)
- mat (aftertouch channel --- ; aftertouch)
- mpb (bendhi bendlo channel --- ; pitchbend)

Type

127 60 0 mkd

to play a middle C (key number 60) on MIDI channel one\(^4\), with a velocity of 127. To release the key, type

0 60 0 mkd

(a velocity of zero means "release the key").

Let's play a note that lasts exactly \(\frac{1}{2}\) second. The word

```
time-advance ( delay --- ; )
```

pauses for the specified delay. The argument to `time-advance` is in units called system virtual time (SVT) whose default length is one millisecond\(^5\). Type the following on a single line:

```
127 60 0 mkd 500 time-advance 0 60 0 mkd
```

When you hit carriage return, you will hear a middle C sound for 500 milliseconds (\(\frac{1}{2}\) second).

FORMULA also provides $words, a more compact notation for playing notes. For example, try

```
c $```

The word `c` pushes the number 60, and the word `$` plays (and releases) a note of the given pitch. In general, the $words should be used in preference to the MIDI primitives given above.

2.2. Defining New Words

You can define note-playing words in FORMULA as follows:

---

\(^3\) A "word" is the Forth equivalent of a function or program. If you are not familiar with Forth, see Appendix B.

\(^4\) The number 0-15 are used to represent MIDI channels 1-16.

\(^5\) The length of an SVT unit can be changed using

```
usecs-per-SVT ( n --- set the length of an SVT unit to n microseconds ).
```

For example,

```
500 usecs-per-SVT
```

sets the SVT unit to 500 microseconds (\(\frac{1}{4}\) millisecond), halving all durations and doubling the tempo. However, there are usually better ways to change tempo; alternatives are discussed in §3.5 and §5.4.
Constructs: \texttt{ap word-name}
\texttt{definition}
\texttt{;ap}

Definitions can be typed in directly or loaded from a disk file (see Appendix B). \texttt{ap} and \texttt{;ap} are exactly like \texttt{:} and \texttt{;} except that they make an additional "vocabulary" visible within the definition (§11.1). The "\texttt{ap}" stands for \textit{active process}, explained below.

For example, type the following:

\begin{verbatim}
:ap devil
  c $ f+ $
;ap
\end{verbatim}

You have defined a word named \texttt{devil} that plays a "devil's interval", C to F#. Type \texttt{devil} to hear it.

Once you have defined a word, you can use it in other words. For example:

\begin{verbatim}
:ap devils
  5 0 do devil loop
;ap
\end{verbatim}

produces five \texttt{devils} in a row. This ability to define new words out of previously defined words can be used to build phrases out of motives, sections out of phrases, and so on.

Like all Forth words, note-playing words can be passed arguments on the stack. Suppose you would like to hear 7 \texttt{devils} in a row, transposed by the notes in a whole tone scale (i.e., C, D, E, F#, G#, A#, C). This can be done most easily by defining a \texttt{transposed-devil} that takes a transposition argument:

\begin{verbatim}
:ap transposed-devil  ( transposition ... ; )
  dup c +$
  f+ +$
;ap
\end{verbatim}

We then call \texttt{transposed-devil} at each transposition desired:

\begin{verbatim}
:ap whole-devil
  12 0 do
    i transposed-devil
  2 +loop
;ap
\end{verbatim}

2.3. Processes

While trying the above examples, you might notice that there is no response to terminal keyboard input while music is playing. To remove this limitation, FORMULA lets you execute multiple \textit{processes} at the same time. This is done using a software technique called \textit{multiprogramming} in which the CPU switches rapidly between processes to provide the illusion of simultaneous execution.

A \textit{process} is a sequential "thread of control". Don't confuse processes and words; several processes can be executing within a single word (perhaps at different points within it) at once. FORMULA uses processes for various purposes (see Figure 1):

- \textit{Note-playing} (or \textit{active}) processes generate streams of notes. You can have several running at once, and you can type Forth commands or edit files with \textit{emacs} while they run.
- \textit{Auxiliary} processes can be used by note-playing processes to generate rhythms, volume changes, tempo changes, and articulation (§5).
- **Input-handling** processes deal with input devices such as the terminal keyboard, the mouse, and MIDI keyboards (§8). The process that handles keyboard input is called the *Forth interpreter*.

  The following word is like *whole-devil*, but runs as a separate process:

  ```forth
  :ap holy-devil
  ::ap whole-devil ;;ap
  ;ap
  ::ap and ;;ap bracket the code to be executed by the new process. In this case the process will execute *whole-devil*, then exit.

  Type *holy-devil*, and you will hear music. You can continue to type while it is playing. By running *holy-devil* a few times, you can get several copies of *holy-devil* to run simultaneously.

  This construct works as follows: the *parent process* (the Forth interpreter, in this case) executes ::ap, which creates a new process (the *child* process) and skips to just past ;;ap. The child process begins executing the code after ::ap. When it reaches ;;ap, it exits and goes away.

  Every process has its own *data stack* for parameter passing and *return stack* for return addresses. Since a parent process and its child have separate stacks, parameters cannot be passed between them in the same way that parameters are passed between words. Instead, the number of

---

Use `control-p` or the ↑ key to do this with minimal typing; see Appendix B.
parameters to be transferred to a new process must be declared within the child process using:

```
[ n params ]
```

where \( n \) is the number of parameters needed. This instructs \::\!ap\!::\) to copy \( n \) words from the parent's stack to the child's stack. For example, the word:

```
:ap tardy-devil    ( delay --- )
::ap
[ 1 params ]
    time-advance whole-devil
::ap
```

will create a process that waits a while before playing.

2.4. Non-Standard Number Formats

Since the Atari ST can't do floating-point arithmetic efficiently, FORMULA uses other representations for fractional quantities, and provides the following notations for these forms:

- Expressions of the form \( A \cdot B \) are converted to their equivalent 8-bit fixed-point values. For example, \( 1.0 \) is converted to 256 and \( 0.5 \) is converted to 128. This is used to specify fractional pitch values (§6.1) and tempos (§5.4).

- Expressions of the form \( A \mid B \) and \( A \cdot B \) represent rational numbers (§3.5) that are used to specify time intervals.

These can be used in both interactive commands and word definitions.
3. MORE FEATURES

3.1. Process Groups

Note-playing processes can be collected into groups. A group can be controlled (suspended, resumed, or killed) as a single unit (§3.3). You can also control the tempo or volume of the notes played by the processes within a group (§5.3 and §5.4). Groups may also contain other groups as elements.

A group is created using the following construct:

::gp
\ ... code for the group's initial process
::gp
\ the original process resumes here when group processes exit

This can be used only within an active process definition. It works as follows: when a process \( P \) executes ::gp, it becomes a group. A new process \( Q \) is created, executing the code following the ::gp, and becomes the sole member of the new group. \( Q \) may create additional processes in the group using ::ap. When all the processes in the group have exited, \( P \) becomes a process again and resumes execution after the ::gp.

For example:

::ap (trio
::gp
::ap soprano ::ap \ create more processes within the group
::ap alto ::ap bass
::gp
::ap
::ap trio \ non-blocking version of trio
::ap (trio ::ap
::ap

The word \( \textit{trio} \) makes the calling process into a group containing three processes (two of these are created by ::ap to run soprano and alto; the third, created by ::gp, runs bass). The calling process resumes only when all three processes have finished. Therefore if you call \( \textit{trio} \) from the Forth interpreter, keyboard input will be ignored until the piece is over. \( \textit{trio} \) is a non-blocking version of \( \textit{trio} \). If called from the interpreter, it creates a new top-level process in which \( \textit{trio} \) is executed.

The ::gp construct also provides a convenient way to wait for a process to finish. Namely, the process that calls ::gp waits for the processes created within the group to finish (there is no easy way to do this using ::ap). In \textit{concert}, for example, this is used to play a sequence of pieces.

3.2. Process Naming

Each process has a corresponding data structure called a \textit{context block} (CB), which contains its stacks and other private data. Internal to FORMULA, a process is identified by the address of its CB. However, since CB addresses are typically 6-digit numbers, this is not convenient at the user-interface level. Instead, a process can have a \textit{small integer identifier} (ID). These ID's are used by words that are \textit{interactive} (i.e., intended to be typed by the user) to refer to processes. By default, a process has no ID; it can request one using

\texttt{assign-proc-ID \ ( --- ; assign an ID to this process )}
In addition, processes can be given symbolic names:

```
proc-name" <string>" ( --- ; assign a name to the calling process )
```

These symbolic names are not used directly to refer to processes, but are printed by \texttt{.all} (see below) to let you associate ID's and processes.

Many \textsc{formula} programs begin by making themselves into a group with a name and ID, as follows:

```
:ap toccata
::ap
    assign-proc-ID \ give it an ID
    proc-name" toccata" \ and a name
    ::gp \ make it into a group

\ create more processes, play music, etc.

    ::gp
    :::ap
::ap
```

By making itself into a separate group, the program is "insulated" from other concurrent activities.

3.3. Process Display and Control

The word

```
.all ( --- ; list all processes that have ID's)
```

shows a summary of the existing processes that have been assigned ID's; other processes are not shown. The following words can then be used to obtain more information about a particular process:

```
.gp ( ID --- ; list all descendants of a process group )
.cb ( ID --- ; print the contents of a process's context block )
.aux ( ID --- ; list a process' auxiliary processes)
.id->cb ( ID --- CB ; convert an ID to the corresponding CB )
(.gp ( CB --- ; list all members of a process group and its descendants)
(.cb ( CB --- ; print the contents of a process's context block )
(.aux ( CB --- ; print a process' auxiliary processes)
```

The following words manipulate note-playing processes and groups. They can be applied only to "top-level" objects, not to elements of groups.

```
suspend ( ID --- ; suspend a process or group )
resume ( ID --- ; resume a suspended process or group )
(suspend ( CB --- ; suspend a process or group )
(resume ( CB --- ; resume a suspended process )
.kill-all ( --- ; kill all processes and groups )
.kill ( ID --- ; kill a process or group )
(kill ( CB --- ; kill a process or group )
.immortal ( --- ; make the calling process immune to kill-all )
```

3.4. Per-Process Storage

\textsc{formula} provides a \texttt{quan} construct for defining named variables:
quan  ( name ; --- ; define a quan )
       ( name ; --- n ; a quan returns its value )
to   ( name ; n --- ; store to a quan )
addr ( name ; --- address ; find address of quan )

For example:

quan counter \ declare a quan named "counter"
li to counter \ store a value in it
counter .    \ get and print the current value

A per-process variable has a different copy for each process (a quan has only one copy,
shared by all processes). Since each process has its own stack, data stored on the stack is
automatically per-process. However, it is not always convenient to refer to stack locations, espe-
cially when they belong to other processes.

FORMULA supports per-process quans, or pquans.

pquan  ( name ; --- ; define a pquan )
pallot ( n --- ; allocate n additional bytes to the last defined pquan )
poffset ( name ; --- n ; find CB offset of pquan )

The to and addr operations for pquans are the same as those for quans; they refer to the cal-
ling process’s copy of the pquan. Likewise, a pquan name by itself returns the value of the cal-
ling process’s copy of the pquan. Other words are provided for accessing other process’s copies
of pquans (see below).

For example:

pquan offset \ declare a pquan
:ap shifty-devil ( transposition --- ; )
::ap
   [ 1 params ]
to offset \ store the transposition in the pquan
  3 0 do
    offset transposed-devil
  loop
::ap
;ap

If several processes execute shifty-devil at once, each will have a separate copy of
offset, and will therefore have an independent transposition.

The following words manipulate the pquans of other processes:

pget  ( name ; CB --- n ; get the value of another process’s pquan )
ipget ( name ; ID --- n ; get the value of another process’s pquan )
pto   ( name ; n CB --- ; store into another process’s pquan )
ipto ( name ; n ID --- ; store into another process’s pquan )
paddr ( name ; CB --- address ; find address of another process’s pquan )
ipaddr ( name ; ID --- address ; find address of another process’s pquan )

For example, if 3 is the ID of a process executing shifty-devil, the command

7 3 ipto offset

will change its offset to 7.

The pquans used internally by FORMULA are initialized at process creation; in most cases
they are inherited from the parent process. User-defined pquans, however, are not automatically
initialized in new processes.
3.5. Specifying Rational Time Intervals

Most musical time intervals can be expressed as a rational number (i.e., ratio of integers) of whole notes. For example, a quarter note is 1/4 of a whole note, and a half note triplet is 1/3. FORMULA lets you specify time intervals as rational numbers. The word $r>i$ converts from rational form to integer (SVT) form. For example:

```
:ap quarter-C
127 60 0 mkd \ key down
1|4 r>i time-advance \ wait a quarter note
0 60 0 mkd \ key up
:ap
```

FORMULA converts expressions of the form $A\div B$ to $A \times B \times r>i$. Hence the above example could be written

```
:ap quarter-C
127 60 0 mkd \ key down
1|4 time-advance \ wait a quarter note
0 60 0 mkd \ key up
:ap
```

The conversion performed by $r>i$ involves an integer computation of the form

$$(\text{num} \ast \text{rscale}) \div \text{denom}$$

where rscale is a pcan representing the SVT duration of a whole note. The default value of rscale is 2000, giving a quarter note the duration

$$(1 \ast 2000) / 4 = 500 \text{ SVT units}.$$  

With the default SVT unit of one millisecond, this would be .5 second (i.e., 120 beats per minute).

Modifying rscale can be used for per-process tempo control for processes that use rational time specification. It can be done by a statement of the form

```
1500 to rscale
```

in the note-playing word, or by an interactive command of the form

```
1500 3 ipto rscale
```

where 3 is the ID of an existing note-playing process. Alternatively, the word

```
beats-per-minute \ ( n --- ; set rscale to give n beats per minute )
```

sets rscale to the value that, at the default SVT unit of 1 millisecond, will produce $n$ quarter-note beats per minute.

3.5.1. Round-off Accumulation

$r>i$ accumulates round-off\(^7\) on a per-process basis. If round-off were ignored, timing errors could occur after repeated rational-to-integer conversions. For example, a sequence of three 1/3-note triplets would be converted to

\[
\frac{(2000 \ast 1)}{3} = 666 \\
\frac{(2000 \ast 1)}{3} = 666 \\
\frac{(2000 \ast 1)}{3} = 666
\]

yielding a total time advance of 1998 instead of 2000. With round-off accumulation, each division produces a fractional remainder of 2/3, yielding the sequence (666, 667, 667).

\(^{7}\) "Truncation" is a more accurate term, since $r>i$ rounds downwards.
Round-off accumulation is appropriate when the rational numbers represent a sequence of concatenated time intervals (e.g., time advances). In some cases, however, a process may want to do conversions that do not represent time advances, and whose round-off should not be accumulated (§5.4 and §5.5). In such cases, the following should be used:

\[
(r>i) \quad (n d \quad i; \ \text{rational conversion with no error accumulation}) \\
A(B) \quad (---; \ \text{equivalent to } A \ B (r>i))
\]

3.6. Vocabularies

FORMULA uses multiple vocabularies to hide words that are of interest only in certain contexts. These vocabularies are:

- **internals** (words not intended for direct user access)
- **loaded** (per-file compilation words -- used for recompile)
- **ap-defs** (words relevant to note-playing process definitions)
- **sh-defs** (words relevant to shape definitions)
- **td-defs** (words relevant to time deformation definitions)
- **sg-defs** (words relevant to sequence generator definitions)

Forthmacs maintains a *search list* of vocabularies to be scanned during word lookup. Words such as `:ap` add the relevant vocabulary to the search list. The following

\[
\text{only forth also definitions}
\]

restores the search list to its default state (forth is Forthmac's main vocabulary). This is often done at the start of source files, and it's often necessary to do it manually after a compilation error.

Beware of the following mistake: if you redefine a word defined in, say, `ap-defs`, and mistakenly put the definition in the `forth` vocabulary, subsequently definitions using `:ap` see the old definition of the word since `ap-defs` precedes `forth` in the search list.
4. $\text{-WORDS: A HIGH-LEVEL NOTE-PLAYING FACILITY}$

As described in §2.1, notes can be played using MIDI primitives and \text{time-advance}, or using a higher-level interface called the $\text{-words}$. The latter facility has several advantages:

- A single word plays a note or group of notes. The word arranges for the note-start command, the \text{time-advance}, and the note-end command.
- The specification of volume, tempo and articulation is "factored out" of the note-playing process. Note parameters are obtained from a combination of pquans (see below) and auxiliary processes (§5).
- The $\text{-words}$ indirectly call lower-level words to start and end notes. Because of this indirectness, programs using the $\text{-words}$ can be synthesizer-independent.

As a preview of the $\text{-words}$, try running the following:

\begin{verbatim}
:ap cadence
  g r a r b r 6$
  c e g +c $4
 ;ap
\end{verbatim}

4.1. Simple $\text{-Words}$

A note-playing process can play notes and rests using

\begin{verbatim}
$   \quad \text{(pitch --- ; play a note)}$
rest \quad \text{(--- ; play a rest)}$
$z$ \quad \text{(pitch --- ; play a note but don't time-advance)}$
$c$ \quad \text{(pitch --- ; play an attenuated note, no time-advance)}$
$s$ \quad \text{(pitch --- ; play a note and its octave)}$
$fe$ \quad \text{(pitch delay --- ; play a note in the future)}$
$fa$ \quad \text{(pitch delay --- ; play a note in the future)}$
\end{verbatim}

$\text{plays a note. Pitch can be specified in a variety of ways (§6). By default, it is a MIDI pitch number (60 = middle C). If pitch is zero a rest occurs; rest is defined as 0 $\text{. The duration of the note or rest is obtained from an auxiliary process called a timing sequence generator (TSG). $\text{ does a time-advance by this amount.}}}$

$z$ \text{ is like $\text{, but the process does not advance in time. Thus a process can play a chord by calling $z$ several times, followed by $\text{. $c$ is like $z$, but the volume is attenuated (§4.4). $s$ is like $\text{, but plays the lower octave as well.}}}$

$fe$ \text{ arranges for a note to be played delay time in the future, using the parameters (volume, etc.) in effect now. $fa$ arranges for a note to be played delay time in the future, using the parameters in effect then.}

4.2. Sustain Pedal

Notes can be controlled by a sustain pedal. If a note is released while its sustain pedal is down, it continues to sound until the pedal is released. The following words manipulate sustain pedals:

\begin{verbatim}
pedon \quad \text{(--- ; lower sustain pedal)}
pedoff \quad \text{(--- ; raise sustain pedal)}
ped \quad \text{(--- ; instantaneously raise and lower sustain pedal)}
pedon$ \quad \text{(--- ; lower sustain pedal and time-advance)}
pedoff$ \quad \text{(--- ; raise sustain pedal and time-advance)}
ped$ \quad \text{(--- ; raise and lower sustain pedal, then time-advance)}
\end{verbatim}
4.3. Synthesizer Output Paradigms

To produce synthesizer output, the $-words indirectly call lower-level output paradigm routines. Different output paradigms can be selected. There are currently three paradigms:

- The synthesizer manager (SM) allocates synthesizer voices based on user-specified note priorities, and uses synthesizer drivers for device independence. Non-standard MIDI (and non-MIDI) synthesizers can be accommodated this way.

- The dumb synthesizer manager (DSM) also uses synthesizer drivers, but does no voice allocation.

- Direct MIDI output (DMO) can be used only for standard MIDI synthesizers.

These paradigms are discussed in more detail in §7.

4.4. Pquans Used by the $-Words

The $-words use several quans and pquans to obtain note parameters. In some cases, their meanings depend on which output paradigm is used. The pquans are all inherited on process creation.

$transpose (quan) is added to all pitches. It is in 1/256-semitone units. For example, to transpose all processes by an octave, type:

12.0 to $transpose

$transpose (pquan) is added to the pitches of notes played by this process. It is in 1/256-semitone units.

$volume (pquan) is added to the volume of notes played by this process (volumes are in the range -127 to 128). Its intended use is to compensate for the particular patch being used, and to balance the volume of different processes.

$cvolume (pquan) is added to the volume of "chord" notes played by this process.

$channel (pquan) is used to decide which synthesizer channel(s) are used for notes played by this process. With DMO, it contains a MIDI channel number (0-15). With DSM, it points to a channel descriptor (§7.2). With SM, it points to a synthesizer descriptor (§7.2).

$patch (pquan) determines the patch (instrument sound) of notes played by this process. It is used only with SM and DSM, and its meaning depends on the synthesizer type being used. Processes that use DMO must change patches explicitly using $DMO-change-patch (§7.3).

$location (pquan) determines the spatial location of notes played by this process. It is used only with SM and DSM, and its meaning is synthesizer-dependent. By convention, 0 = left, 64 = center, and 127 = right.

$priority (pquan) is the priority of notes played by this process. It is used only with SM (§7.5). This priority determines which notes to preempt when all voices are used.

$pedal (pquan) selects a logical sustain pedals to be used for notes played by this process. It is used only with SM (§7.5).

4.5. Compound $-Words

To simplify notation, FORMULA allows pitches indices to be grouped together into lists (sequences) and groups (chords) that can then be played with the following compound $-words:
m$ (pitch-list x --- ; play a sequence of x notes)
$n (pitch-group x --- ; play a chord with x notes)
m$n (pitch-group-list x y --- ; play a sequence of x y-note chords)
m$$ (pitch-list n --- ; play n notes with lower octaves)

In each case, the process does a time-advance by the TSG amount after each chord or note is played. For example,

c e g +c 4 m$ (plays a C major arpeggio)
c e g +c 4 $n (plays a C major chord)
c e g +c f a +c +f c e g +c 3 4 m$n (plays a C-F-C chord sequence)

The $-words $, m$, $n and m$n have "iterator" versions formed by appending *k to the word, and supplying the iteration count as the final argument. For example,

c e g +c f a +c +f c e g +c 3 4 5 m$n*k

plays the C-F-C sequence 5 times. This is equivalent to enclosing the whole thing in a do loop, except that pitches are computed just once.

It is convenient to define special versions of the compound $-words with built-in constants. $4 is defined as 4 $n, 8$ as 8 m$, and 4$3*6 as 4 3 6 m$n*k. 4$3*6, for example plays a sequence of 4 3-note chords 6 times. Many commonly-occurring abbreviations of this sort are available (see the files notation and notate2), and others can be defined as needed.

Rolled chords can be played using

$nroll (pitch-group dt n --- ; play n notes spread over time dt)

This advances the process by TSG amount, NOT by dt. Hence the notes in the rolled chord can overlap with the subsequent notes generated by the process.
5. AUXILIARY PROCESSES

5.1. Introduction

The $-words take only pitch arguments. Other parameters (such as volume and duration) are obtained from pquns ($4.4) and from auxiliary processes. Auxiliary processes can also be used to change the tempo of note-playing process. An auxiliary process can be attached to either a process or a group. In the latter case, it affects all processes in group and its descendants. Auxiliary processes allow you to separate musical “interpretation” from the musical “score”.

Auxiliary processes can be created using embedded process definitions. Here is an example, taken from the file pieces/demo:

```latex
:ap (demo
  ::gp
  92 beats-per-minute \ volume control process
  ::gshl
  begin p f 3|4 oseg f p l|1 oseg again
  ::sh
  ::gtdl \ tempo control process
  begin 0.8 1.2 l|1 seg 1|16 lpause 1.2 0.8 5|4 seg again
  ::td
  ::ap \ first note-playing process
  begin
    /16 20 irnd 50 + \ play a random note a random number of times
    brnd 1+ 4 * 0 do dup $ loop drop
    /1 rest
    again
  ::ap
  ::tsg \ rhythm generator for 2nd process
  begin /8+ /4-3 again
  ::sg
  0 begin \ second note-playing process
  20 irnd swap \ uses random/cyclic algorithm
  20 0 do
    over + 40 mod dup 40 + $ loop
    drop
    again
  ::gp
;ap

:ap demo
  ::ap
  assign-proc-ID proc-name" demo"
  (demo
  ::ap
;ap
```

Auxiliary processes can also be created from the interpreter and attached to existing note-playing processes or groups.

5.1.1. Embedded Auxiliary Process Definitions

A note-playing process has local and global contexts, each of which is either the process itself or a group that contains it. By default, a process’s local context is itself, and its global context is the top-level group containing it (or, if the process is top-level, the process itself). The following words can be used to change the contexts:
raise-local-context { --- ; move local context up one level }
lower-global-context { --- ; note global context down one level }

Each object contains slots for auxiliary processes. A slot is a queue that is either zero or points to the CB of an auxiliary process. A note-playing word can contain embedded auxiliary process definitions that create new processes in its slots or those of its local or global contexts. For example:

```lisp
:ap foo
  ::shl \ install new volume shape in local context
    p f d/1 oseg \ code of volume shape process
  ::sh
    c e d f 4$ \ play some notes
  :ap
```

The semantics are as follows: when a note-playing process reaches the start of the definition (::shl in this case) the auxiliary process currently in the shl slot of its local context is killed. A new process is created executing the embedded code, and is installed in the shl slot of its local context. If desired, parameters can be passed to the new process using [ _ n params ].

The constructs for embedded auxiliary process definitions are:

<table>
<thead>
<tr>
<th>syntax</th>
<th>location</th>
<th>purpose</th>
<th>process type</th>
</tr>
</thead>
<tbody>
<tr>
<td>::tsg ... ::sg</td>
<td>self</td>
<td>note duration</td>
<td>sequence generator</td>
</tr>
<tr>
<td>::shl ... ::sh</td>
<td>local context</td>
<td>volume control</td>
<td>shape</td>
</tr>
<tr>
<td>::sh2 ... ::sh</td>
<td>local context</td>
<td>volume control</td>
<td>shape</td>
</tr>
<tr>
<td>::gsh1 ... ::sh</td>
<td>global context</td>
<td>volume control</td>
<td>shape</td>
</tr>
<tr>
<td>::gsh2 ... ::sh</td>
<td>global context</td>
<td>volume control</td>
<td>shape</td>
</tr>
<tr>
<td>::ash ... ::sh</td>
<td>local context</td>
<td>articulation</td>
<td>shape</td>
</tr>
<tr>
<td>::td1 ... ::td</td>
<td>local context</td>
<td>tempo control</td>
<td>time deformation</td>
</tr>
<tr>
<td>::td2 ... ::td</td>
<td>local context</td>
<td>tempo control</td>
<td>time deformation</td>
</tr>
<tr>
<td>::gttd1 ... ::td</td>
<td>global context</td>
<td>tempo control</td>
<td>time deformation</td>
</tr>
<tr>
<td>::gttd2 ... ::td</td>
<td>global context</td>
<td>tempo control</td>
<td>time deformation</td>
</tr>
</tbody>
</table>

5.1.2. External Creation and Deletion of Auxiliary Processes

An auxiliary process can be attached to an existing object using

```lisp
ishl ( name ; ID --- ; )
nish2 ( name ; ID --- ; )
ittdl ( name ; ID --- ; )
ittd2 ( name ; ID --- ; )
itsg ( name ; ID --- ; )
iasl ( name ; ID --- ; )
```

For example

```lisp
3 ishl foo
```
creates a volume shape executing foo and installs it in the shl slot of the object with ID 3 (it is attached to the object itself, not to its local or global context). If name is noop then the slot is cleared. Parameters cannot be passed to the shape.

The auxiliary processes attached to a process can be removed using

```lisp
clear-aux ( ID --- ; )
(clear-aux ( CB --- ; )
```
5.2. Using Processes to Define Functions of Time

Shapes and time deformations define functions of time. Instead of defining functions in the usual way (by taking a time argument and returning a value) they use procedural concatenation. In this style of definition, a function is generated by a process. The process invokes a series of primitives (procedures representing a function defined on an interval) and thereby defines a function that is the concatenation of the primitives. The process may take parameters, and may execute arbitrary code between invocations of primitives.

As an example of procedural concatenation, suppose

\[
\text{oseg} (\ y_1 \ y_2 \ dt \ -- \ ; )
\]

is a primitive representing a linear function varying from \( y_1 \) to \( y_2 \) over a time span of \( dt \). The following function could then be defined:

\[
: \text{sawtooth} (\ n \ -- \ ; \ \text{sawtooth with} \ n \ \text{skewed teeth} )
\]

\[
0 \ \text{do}
\]

\[
0 \ 3 \ 10 \ \text{oseg}
\]

\[
3 \ 0 \ 5 \ \text{oseg}
\]

\text{loop}

;\]

The word \( \text{sawtooth} \) invoked at time zero with argument 2 defines the function shown below:

![Graph of sawtooth function](image)

5.3. Volume Control

Shapes are functions, defined by procedural concatenation, used to control volume and articulation. Their definitions are delimited by \( : \text{sh} \) and \( ; \text{sh} \). The following primitives are currently available:

\[
\text{oseg} (\ y_1 \ y_2 \ dt \ -- \ ; \ \text{linear segment open on right} )
\]

\[
\text{cseg} (\ y_1 \ y_2 \ dt \ -- \ ; \ \text{linear segment closed on right} )
\]

\[
\text{ocon} (\ y \ dt \ -- \ ; \ \text{constant segment open on right} )
\]

\[
\text{ccon} (\ y \ dt \ -- \ ; \ \text{constant segment closed on right} )
\]

\[
\text{inf-con} (\ y \ -- \ ; \ \text{infinite constant-valued segment} )
\]

The closure is relevant for discontinuous functions; it determines the function value at the points of discontinuity. A primitive's closure on the left is determined by the right closure of the preceding primitive.

The volume of a note is represented by a number in the range -128 to 127. The volume of notes played using \$\text{-}\text{-}\$-words is the sum of several components:

- Up to two \textit{local} volume shapes.
- Up to two \textit{global} volume shapes.
• The contents of a per-process variable $v_{volume}$.
• If the note is played with $c\$, the contents of a per-process variable $c_{volume}$.

To avoid overflow, it is useful to limit the contribution that each component can make. The convention is that the sum of all four shapes must lie between -96 and 95, $v_{volume}$ must lie between 0 and 32, and $c_{volume}$ must lie between -32 and 0.

Shape definitions may use the words $ppp$, $pp$, $p$, $mp$, $mf$, $f$, $ff$, and $fff$. Because the corresponding values are synthesizer-dependent, these words are defined in a synthesizer-specific configuration file (§7.2).

As an example, a shape definition such as:

:sh ramp ( 1-measure crescendo )
  p f 1|1 oseg
;sh

could be used in another shape definition:

:sh 10ramps
  10 0 do ramp loop
;sh

or attached to a running note-playing process (process 2 in this example):

2 ish1 ramp

or invoked from a note-playing process

:ap foo ( n --- ; play 100 notes with n crescendi )
  ::shl
    [ 1 params ]
    0 do ramp loop
  ;;sh
    100 0 do c $ loop
  ;;sh
;ap

The last example could also be written as:

:ap foo ( n --- ; play 100 notes with n crescendi )
  ::shl
    [ 1 params ]
    0 do
      p f 1|1 oseg
      loop
    ;;sh
    100 0 do c $ loop
  ;;sh
;ap

5.4. Tempo and Rubato Control using Time Deformations

Continuous tempo fluctuations (e.g., for rubato) are done using time deformations (TD’s).
A TD defines a tempo function\(^8\) by procedural concatenation. A TD is applied to a time interval by integrating the tempo function over the interval, starting from its current time position. For example, if the tempo varies linearly from 1 to 2 over an interval of duration 1, the interval is mapped by the TD to a duration of 1.5 (see Figure 2).

\(^8\) "Inverse tempo functions" might be a better term, since a larger value means a slower tempo.
Figure 2: A time deformation (TD) viewed as (a) a procedural definition, (b) a tempo function that is integrated over its domain, and (c) a process that deforms time intervals.

The following TD primitives exist:

seg ( r1 r2 dt --- ; linear tempo change from r1 to r2 over time dt )
con ( r dt --- ; constant tempo of r over time dt )
inf-con ( r --- ; infinite constant segment )
lpause ( t --- ; pause of t before events )
rpause ( t --- ; pause of t after events )
con.outer ( r dt --- ; like con, but dt is in deformed units )
seg.outer ( r1 r2 dt --- ; like seg, but dt is in deformed units )

seg and con represent linear and constant tempo functions. lpause and rpause insert a "pause" in the tempo function. Events scheduled for this instant occur after the pause with lpause, before it with rpause. FORMULA represents tempi using fixed-point numbers with an 8-bit fractional part; 256 is unity tempo. This allows A.B notation to be used (§2.4).

For example, the following TD does an accelerando over each 4-beat measure, and inserts a slight pause before the start of the next measure:

:td foo
begin
  1.3 .9 1/1 seg
  1/32 lpause
  again
:td

It can then be attached to the note-playing process with ID 2:
2 itdl foo

or written as an embedded definition:

```
:ap blah
  :tdl
  begin
    1.7 .9 1|1 seg
    1|32 lpause
    again
  ;;td
  100 0 do c $ loop
:ap
```

If two TD's are attached to the same object, their effects are multiplied. If TD's are attached to an object and its parent, they are combined in series: the output of one becomes the input of the other.

5.5. Articulation Control

In general the "delay until release" of a note (denoted $D(r)$) can differ from its "delay until next note" (denoted $D(n)$). $D(r)$ may be longer (causing note overlap or legato) or shorter (portamento or staccato) than $D(n)$. This timing relationship will be called articulation. In FORMULA, the articulation for each note-playing process is controlled by an articulation shape that determines $D(r)$ as a function of $D(n)$.

Articulation shapes define both a numeric value and a mode: absolute, relative, or ratio. The mode and the value $X$ determine $D(r)$ as a function of $D(n)$, as follows:

- **absolute**: $D(r) = X$
- **relative**: $D(r) = \text{max}(0, D(n) + X)$
- **ratio**: $D(r) = (D(n) * X)/256$

In other words, the value of an articulation gives either the release time of notes, the release time relative to the start of the next note, or the release time as a multiple (with 8-bit fractional part) of the time until the next note, depending on the mode.

Articulation shapes use the shape primitives (§5.3) to define their numeric value, and the words **absolute**, **relative** and **ratio** to set their mode. For example, the following articulation shape varies continuously from staccato to legato every 4 beats:

```
:sh art-shape
  ratio
  begin
    .1 1.2 1|1 oseg
    again
  ;;sh
```

It can be attached to a running process:

```
2 art-shape iash
```

or written as an embedded process:
5.6. Timing Sequence Generators

A timing sequence generator (TSG) is a process that generates a sequence of note durations for a note-playing process. The $-$-words get note durations from TSG's.

TSG word definitions are delimited by :sg and ;sg. Embedded definitions are delimited by ::sg and ::sg. A TSG generates a single sequence element using

```
& ( n --- ; return a sequence element )
```

The value returned by & may be generated using any of the time specifications described in the previous section. A set of commonly occurring rhythmic patterns are defined using the following naming conventions:

<table>
<thead>
<tr>
<th>name</th>
<th>definition</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>/1</td>
<td>1 1 &amp;</td>
<td>whole note</td>
</tr>
<tr>
<td>/4</td>
<td>3 1 8 &amp;</td>
<td>dotted quarter</td>
</tr>
<tr>
<td>/4..</td>
<td>7 16 &amp;</td>
<td>double dotted quarter</td>
</tr>
<tr>
<td>/4,,</td>
<td>5 16 &amp;</td>
<td>quarter plus a 16th</td>
</tr>
<tr>
<td>/4,,,</td>
<td>9 32 &amp;</td>
<td>quarter plus a 32nd</td>
</tr>
<tr>
<td>/8, etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2/8</td>
<td>/8 /8</td>
<td>two eighths</td>
</tr>
<tr>
<td>16/32 etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/4.8</td>
<td>/4. /8</td>
<td>dotted quarter and eighth</td>
</tr>
<tr>
<td>/2.4 etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/2-3</td>
<td>/2/3 /2/3 /2/3</td>
<td>triplet half</td>
</tr>
<tr>
<td>/4-3 etc.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/8+</td>
<td>/8 /16 /16</td>
<td>eighth and two 16ths</td>
</tr>
<tr>
<td>/4+ etc.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above words are in the sg-defs vocabulary since they are called only from TSG definitions. The names /1, /2, /4, /8, /16, /32, /4.8, /2.4, /2+, /4+, /8+, /4-3, and /8-3 are also used in the ap-defs vocabulary for words that install a TSG that generates an infinite sequence of the given durations. This allows the following convenient notation in note-playing processes:

```
:ap foo
/4 e 3:5 \ play some quarter notes
/8 f g a 3:5 \ now play some eighth notes
:ap
```

An example of a TSG definition:
:sg funky
 2/4 /4+  \ three quarters and two eighths
 5 0 do 8/8 /2-3 loop  \ eight eighths and a triplet half, repeated five times
\ :sg

funky could be used in another TSG definition:
  :sg blah
  funky 4/8 funky
\ :sg

This defines a sequence composed of 2 copies of the funky sequence with 4 eighths interposed.

funky could be used in a note-playing process:
  :ap player
   ::tsg funky ;;sg \ start out in funky rhythm
   8 0 do c $ d $ loop \ play some notes
   /16
   8 0 do c $ loop \ switch to sixteenths and play more notes
\ :ap

This example could be rewritten as an embedded TSG definition:
  :ap player
   ::tsg
     2/4 /4+ \ define funky rhythm
     5 0 do 8/8 /2-3 loop
   ;;sg
     8 0 do c $ d $ loop \ play some notes using funky rhythm
     /16
     8 0 do c $ loop \ switch to sixteenths and play more notes
\ :ap
6. PITCH SPECIFICATION

6.1. Pitch Values and Pitch Indices

FORMULA provides several ways of specifying pitches. Two numeric representations are used:

- A *pitch value* is a fixed-point number with 8 fractional bits, representing a number of semitones\(^9\). 57.0 (represented as 57*256) is 440 Hz.
- A *pitch index* is an integer used to represent an element of a discrete scale.

The $-words normally take a pitch index as an argument. FORMULA provides two high-level mechanisms for specifying pitch indices: *symbolic pitch names* (a, b, ...) and *pitch sets* for specifying scales or chords. Both are described later in this section.

A `p`quan `$pitch-convert` points to a word, called automatically by the $-words, that converts from pitch index to pitch value. FORMULA supplies several alternative conversion words:

- `shift-convert` multiplies its argument by 256. Pitch indices then refer to the 12-tone equal tempered scale (this is the default.)
- `null-convert` is a no-op; this lets you pass pitch values directly to the $-words.
- `tuning-convert` maps the pitch index into an element of a scale defined by the *tuning system* facility described below.

For example:

```
' null-convert to $pitch-convert 60.5 $ \ play a quartertone above middle C
' shift-convert to $pitch-convert 60 $ \ back to equal temperament \ play middle C
```

6.2. Symbolic Pitch Names

The words a, b, ..., g push a pitch index on the stack. The octave is not explicitly given with the pitch name. Instead, each process has a *current octave*, and a, b, ..., g refer to the note instance in this octave. Octaves range from C flat up to B sharp. Middle C is in octave 3, which is the initial octave of all processes. The current octave can be changed using

```
oct     ( n --- ; set current octave to n )
+oct     ( --- ; increment current octave )
-oct     ( --- ; decrement current octave )
```

Sharps and flats are denoted by appending + or - suffix (f+, b- etc.). Prepending a + or - to selects a pitch one octave above or below normal. A number (zero) representing a rest is pushed by the word r.

6.3. Tuning Systems

FORMULA includes a facility for defining and using nonstandard tuning systems (justintoned scales, stretched tunings, etc). The facility is set up to be used with the $-words and the synthesizer manager; with a little work it could also be used independently. When used, each key-down command that the SM dispatches to a synthesizer driver includes a pitch value expressed as a multiple of 1/256 of a semitone. Synthesizer drivers use the fractional part if they

\(^9\) Of course, the fractional bits are relevant only for synthesizers with microtonal capabilities.
can\textsuperscript{10}.

An example of a tuning system definition:
\begin{verbatim}
\verbatim{\textbackslash make an array of pitch value offsets}
create \{just 70 p, 182 p, 275 p, 386 p, 498 p, 569 p, 702 p, 773 p, 884 p, 996 p, 1088 p,}
\verbatim{\textbackslash declare a scale named "just" using these offsets}
scale: just
12, 1206 p, (just ,
\end{verbatim}

The first part creates a list named \texttt{(just of fractional pitch offsets}. The word \texttt{p}, converts an offset expressed in cents \texttt{(1/100 of a semitone)} into an offset in \texttt{1/256 of a semitone}\textsuperscript{11}. The second part defines a tuning system named \texttt{just}. It repeats every 12 pitches, and it ascends an octave plus 6 cents every period.

The word \texttt{just} is called with a pitch index \texttt{n} as an argument. This causes the scale to be used in the calling process. The \texttt{“origin”} of the scale is \texttt{n}: the pitch index \texttt{n} is mapped to its equal-tempered pitch value, and other pitch indices are mapped according to the tuning system. For example,
\begin{verbatim}
c just
\end{verbatim}
causes the calling process to play in a just-intonated scale starting at C. Other processes can use different tuning systems.

The following tuning systems are predefined (see the file \texttt{scales})\textsuperscript{12}:
\begin{verbatim}
stretch
just
pent
pelog-barang
\end{verbatim}
\texttt{stretch} is 12-tone equal temperament with each octave stretched by 8 cents. \texttt{just} is a 12-tone just-intonated scale. \texttt{pent} is a just-intonated pentatonic scale. It has been defined as a 12-note scale where groups of 2 and 3 indices map to the same pitch. \texttt{pelog-barang} is a 5-note Gamelan scale.

6.4. Pitch Sets

FORMULA provides a \texttt{pitch set} facility for defining scales, chords, and other arbitrary ordered groups of pitch indices. As shown below, this facility maps a \texttt{pitch set index} (i.e., scale degree) into a pitch index, which may then be mapped to a pitch value by the tuning-system mechanism described earlier.

\textsuperscript{10} The facility is geared towards synthesizers that can be sent an independent pitch per note (such as the Yamaha FB-01) rather than those that require tunings to be downloaded into the synthesizer (such as the Yamaha DX7-II and TX81Z).

\textsuperscript{11} The pitch offset in cents corresponding to a frequency ratio of \texttt{n/m} is \texttt{1200*\log(n/m)/\log(2)}. This formula can be used to enter a scale defined in terms of rational intervals.

\textsuperscript{12} The manual for the \texttt{“Tune Up”} program (written by Tim Perkis, published by Antelope Engineering, 1048 Neilson St., Albany, CA 94706) is a rich source of data for nonstandard scales.
A pitch set may consist of separate ascending and descending sets of pitch indices. Each of these has an associated pitch offset table, a list of pitch-index offsets of the elements of the set. Each set is invariant under translation by a pitch set period (usually 12). A pitch set is described by a pitch set template and an origin. A pitch set template has four items:

- The number of entries in each pitch offset table.
- The pitch set period.
- A pointer to the pitch offset table for the ascending group.
- A pointer to the pitch offset table for the descending group.

Elements of the (ascending) pitch set consist of the origin, plus an element of the (ascending) pitch offset table, plus an integral multiple of the pitch set period.

The file `pitchset` contains definitions of several popular chords and scales. For example, here is the definition and use of a melodic minor scale:

```plaintext
\first {define pitch offset tables
create ascmin 0, 2, 3, 5, 7, 9, 11,
create descmin 0, 2, 3, 5, 7, 8, 10,}
\now {define pitch set template
create minorscale 7, 12, ascmin, descmin,}
:ap foo \play ascending and descending minor scales
g minorscale set-ps
20 0 do +ps $ loop
20 0 do -ps $ loop
;ap
```

For each process, the current PS and position within PS are maintained. The following words initialize pitch set use and convert pitch set indices to pitch indices.

- `set-ps` (origin template-address; switch to new PS with the given origin and template)
- `+ps` (--- pitch; next note up in ascending PS)
- `-ps` (--- pitch; next note down in descending PS)
- `+nps` (n --- pitch; n steps up in ascending PS)
- `-nps` (n --- pitch; n steps down in descending PS)
- `aps` (n --- pitch; nth element of ascending PS relative to origin)
- `dps` (n --- pitch; nth element of descending PS relative to origin)
- `psind` (--- n; a pquan that stores last PS index used)
- `pslast` (--- pitch; a pquan that stores last pitch returned)
7. SYNTHESIZER OUTPUT

7.1. Output Paradigms

The capabilities of the synthesizers FORMULA controls will vary from user to user. For example, some synthesizers can play several notes at once, but are controlled using a single MIDI channel; we call these 1-channel synthesizers. Others can receive commands on multiple MIDI channels simultaneously, and can play at most one note on a channel at once; we call these n-channel synthesizers (they are also called multitimbral because each channel can produce a different instrument sound). A user may have several synthesizers of different types, or may have synthesizers with non-MIDI interfaces (e.g., using the Atari's parallel port). A synthesizer configuration is the complete set of synthesizers being controlled by FORMULA.

After a $-word obtains a note's pitch, volume, and other parameters, how does it actually play the note? The answer is supplied by an output paradigm, a software layer interposed between $-words and synthesizers\textsuperscript{13}. FORMULA provides three output paradigms:

- Direct MIDI output (DMO) directly generates MIDI commands. The program must specify which MIDI channel(s) to use.
- The dumb synthesizer manager (DSM) provides synthesizer independence. Instead of generating output directly, it calls synthesizer drivers to do the actual output (these drivers are supplied as part of the description of the configuration). This lets you use non-standard features of MIDI synthesizers (and non-MIDI synthesizers) with no program modifications. Programs still must specify the "channel" (not necessarily a MIDI channel) to be used.
- The synthesizer manager (SM) provides device independence and also does synthesizer voice allocation. Programs don't tell it which channel to use. Instead, the SM allocates channels (and voices within channels) dynamically, based on note priorities supplied by the program. For example, this can be used to ensure that melody notes are not preempted by the accompaniment.

The following diagram shows the various software components involved with synthesizer output:

\textsuperscript{13} Processes may also access output paradigms directly, without using $-words. This interface is not described here, but can be found in the source code.
7.2. Synthesizer Configuration

Both the SM and the DSM need to be told about your synthesizer configuration: how many channels of each synthesizer type, and how many voices per channel. This is done using

\[
\begin{align*}
\text{declare-synth} & \quad \text{(--- synth-desc; declare a synthesizer type)} \\
\text{declare-channel} & \quad \text{(channel-no synth-desc driver-addr nvoices} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{--- channel-desc; declare a synthesizer channel)}
\end{align*}
\]

declare-synth returns a synthesizer descriptor that identifies a synthesizer type\(^{14}\). The arguments to declare-channel are as follows. synth-desc is the synthesizer type of the channel, and driver-addr is the address of a synthesizer driver routine (see below). When the driver is called, channel-no is passed to identify the channel; for MIDI, this is the MIDI channel number. nvoices is the number of voices (i.e., maximum number of simultaneous notes) on the channel.

For example, here's a configuration routine for two MIDI synthesizers, a Yamaha with 1 voice each on MIDI channels 1-8, and a Roland with 16 voices on channel 10.

\(^{14}\) A "synthesizer type" is really just the set of channels of that type.
quan yamaha-sd \ points to synthesizer descriptor for Yamaha
quan roland-sd \ points to synthesizer descriptor for Roland

: my-synth-config
declare-synth to yamaha-sd \ create a synthesizer descriptor for Yamaha
declare-synth to roland-sd \ create a synthesizer descriptor for Roland

\ declare the 8 Yamaha channels
8 0 do
  i yamaha-sd [\'] generic-MIDI-driver 1 declare-channel
drop \ the SM doesn't need the channel descriptor, so drop it
loop

\ declare the one Roland channel
9 roland-sd [\'] generic-MIDI-driver 16 declare-channel
drop
;

In this example, generic-MIDI-driver is a synthesizer driver routine that generates standard MIDI commands\textsuperscript{15}.

7.2.1. Selecting an Output Paradigm

A process "selects" an output paradigm by setting a pair of pquans ($note-routine and $pedal-routine) to point to routines provided by the paradigm. The words $SM, $DSM and $DMO set up these pquans for the paradigms described above. Because pquans are used, different processes may use different output paradigms\textsuperscript{16}. A process inherits these pquans from its parent, as well as the other pquans used by the $-words ($channel, $patch, $priority, etc.).

For example, the following routine selects the SM output paradigm and arranges to use the Yamaha synthesizer (as declared above) with patch zero.

: my-select-paradigm
$SM
  yamaha-sd to $channel
  0 to $patch \ initial patch (Brass?)
  64 to $location \ initial stereo location (center)
\ other pquans ($priority, $pedal, etc.) are initially zero
;

7.2.2. Initialization

The initialization word \texttt{formula} calls two deferred words:

- \texttt{set-synth-config} declares your synthesizer configuration to the SM.
- \texttt{select-paradigm} connects the $-words to an output paradigm. This involves initializing the pquans $note-routine and $pedal-routine and initializing other pquans (such as $channel) used by the paradigm.

You can redefine these; for example,

\textsuperscript{15} FORMULA also includes a driver \texttt{fb01-driver} for the Yamaha FB-01 that supports microtonality using the FB-01’s system exclusive messages.

\textsuperscript{16} However, unexpected results may occur if a particular synthesizer is accessed via two different paradigms.
my-synth-config is set-synth-config
my-select-paradigm is select-paradigm

causes formula to select the SM and declare the synthesizer configuration given above. If this is done in a configuration file that is loaded prior to saving formula.tos, then this will be done automatically each time you run formula.

The default version of select-paradigm selects DMO, and the default version of set-synth-config is a no-op.

7.3. Direct MIDI Output

When direct MIDI output (DMO) is used, notes played with the $-words are translated directly into standard MIDI commands. There is no voice allocation, and no synthesizer drivers.

A configuration file for using DMO looks like this:

: select-DMO
  $DMO \ initialize $note-routine, $pedal-routine
  0 to $channel ; \ use MIDI channel zero
  
select-DMO is select-paradigm

DMO uses the $channel pquan to hold a MIDI channel number, which is included in all commands sent. DMO does not use the $patch pquan to select a patch. Instead, note-playing processes must use an explicit command:

DMO-change-patch ( new-patch --- ; )

Therefore an instrument definition word for DMO might look like

: ap piano
  4 DMO-change-patch
  10 to $volume
  12.0 to $transpose
  ; ap

7.4. The Dumb Synthesizer Manager

The DSM provides synthesizer independence. The pquan $channel must be set to the channel descriptor of the synthesizer channel to be used. Therefore configuration routines must put channel descriptors in well-known places (e.g., quans). For example, the following configuration file is for a Yamaha FB-01 set up as 8 voices on channel 0, to be accessed via the DSM:

quau fb01-sd \ synth descriptor for FB-01
quau fb01-cd \ channel descriptor for FB-01's channel zero

: fb01-config-synth
  declare-synth to fb01-sd
  0 fb01-sd [''] fb01-driver 8 declare-channel to fb01-cd ;

: fb01-select-DSM
  DSM
  fb01-cd to $channel
  0 to $patch
  64 to $location ;

' fb01-config-synth is config-synth
' fb01-select-DSM is select-paradigm
A process that uses DSM does not schedule an event to change instruments; it merely changes its $patch and $location pquans (unlike DMO, no explicit call is needed). Hence one might define words like

```
:ap piano
  th 304 to $patch
  10 to $volume
  12.0 to $transpose \ commentate for soft patch
  \ compensate for low patch (units are 1/256 semitone)
:end

:ap $center
  64 to $location
:end
```

When invoked from a process, piano will cause subsequent notes to be played using those parameters. The contents of $patch depend on the synthesizer type; they are interpreted by the synthesizer driver. In this case (for the FB-01) they include a bank number (3) and a patch number in that bank (4).

Using DSM with the generic MIDI driver is similar to using DMO, but there is one major difference. Suppose two processes use the same MIDI channel and change patches periodically. With DMO, each process will (perhaps unwantedly) be affected by the change-patch commands generated by the other. With DSM, change-patch commands are automatically generated for each note, as needed; all notes will be played with the correct patch.

### 7.5. The Synthesizer Manager

When using the SM, the pquan $channel must point to the descriptor for the synthesizer type to be used\(^1\). The $priority pquan specifies a priority for notes played by the process. The SM chooses the "best" channel to use for each note, and decides which existing note(s) to preempt. The SM uses the channel for which the priority of preempted notes is lowest. Processes that are playing crucial parts (e.g., melodies) should store a high value in $priority.

In addition, each note is assigned a pedal group, numbered 0 through 15, using the $pedal pquan. These "logical sustain pedals" can be manipulated independently; the pedal-manipulation words (§4.2) affect only the pedal specified by $pedal. When a logical sustain pedal is raised or lowered, all the notes using that pedal are affected. These notes may be distributed over several channels, and even over different synthesizer types.

As with the DSM, instrument parameters are controlled by changing the $patch and $location pquans.

### 7.6. The MIDI Output Driver

FORMULA's MIDI output driver is a collection of words, assembly language macros, and interrupt handlers. To improve performance, the driver eliminates redundant command bytes. It remembers the current "MIDI state", i.e., the last command byte output on the MIDI line. If the next command has the same command byte, the command byte is suppressed and only the data bytes are sent. For robustness, this mechanism is overridden (i.e., the command byte is sent regardless of the MIDI state) if no command byte has been sent in the last 1/2 second.

This concept is extended further to accommodate synthesizers with a system-exclusive event

---

\(^1\) The $channel pquan is slightly overloaded; it points to a synthesizer descriptor in this case, not a channel descriptor.
Such states are entered by a multi-byte system exclusive command, and exited by a $7$ byte. FORMULA's MIDI output driver represents each system-exclusive state by an index into an array `sysexc1-table`. Each element of this array points to a MIDI string (starting with a 32-bit length field) used to enter the corresponding state. A synthesizer driver that uses system-exclusive states must allocate an entry in this array at compile time (using `next-sysexc1-index`) and must pass the index to the MIDI output driver as the "command byte" of commands in its system-exclusive state.

MIDI channel states are encoded as follows: values from 128 to 257 represent standard MIDI command states, and values from 1 to 127 represent system-exclusive states corresponding to the entries in `sysexc1-table`. 0 represents no state (i.e., the state initially or immediately after a system-exclusive command).

The MIDI output driver can be accessed in several ways:

- **MIDI-command**
  
  (Dn ... D2 command n --- ; output a MIDI command)

- **MIDI-output**
  
  (Dn ... D1 n --- ; output raw MIDI bytes)

- **MIDI-command-event**
  
  (machine-code event routine version of MIDI-command)

- **MIDI-output-event**
  
  (machine-code event routine version of MIDI-output)

- **(MIDI-command)**
  
  (assembler macro version of MIDI-command)

- **(MIDI-output)**
  
  (assembler macro version of MIDI-output)

*MIDI-command* and *MIDI-output* are high-level words, and may be called only within `::ev` and `;::ev`\(^\text{19}\). Both words take an argument count \(n\) and a sequence of \(n\) bytes (unpacked, 1 per 32-bit stack entry). *MIDI-command* interprets the bottom byte as a MIDI command state, as described above, and suppresses it if possible. *MIDI-output* simply outputs the bytes without interpretation.

*MIDI-command-event* and *MIDI-output-event* are machine-code event routines with the same arguments, to be scheduled using `event` or `future-event`. *(MIDI-command)* and *(MIDI-output)* are assembly-language macros, to be used in synthesizer driver routines. They expect a pointer to an event record (§10.1) in A1. The `nargs` and `args` field of the event record contain the MIDI command. They do not free the event record; hence synthesizer drivers can use a statically-allocated record.

### 7.7. How to Write a Synthesizer Driver

A **synthesizer driver** is a machine-code word for controlling a particular type of synthesizer. The driver is passed detailed information about notes: stereo location, microtonality, and so on. It uses as much of the information as it can, given the capabilities of the synthesizer.

A synthesizer driver is called by `jsr` from the SM or DSM, and must return by `rts`. Its arguments are passed in registers:

- **d0:** command code (see below)
- **a4:** pointer to a channel descriptor (all commands)
- **a3:** pointer to a note descriptor (key up/down command only)
- **d3:** additional argument (all commands except key up/down)

The routine is allowed to change only d0-d2 and a0-a2. The command codes are:

\[
\begin{align*}
0 &= \text{key up/down (the note and channel descriptors contain the pitch, volume, channel number, etc.)}
\end{align*}
\]

\(^{18}\) For example, the Yamaha FB-01 has a system-exclusive state in which key-down commands specify fractional pitches.

\(^{19}\) *MIDI-command* and *MIDI-output* don't mask interrupts, so there are potential synchronization problems if you call them directly. Event performance, however, is automatically sequential.
1 = change patch (the new patch number is passed in d3).
2 = sustain pedal (an up/down flag is passed in d3; pedal down if nonzero).
3 = parameter change (the index of the parameter is passed in the high word of d3, and its
value is passed in the low word of d3). Parameter indices are synthesizer-dependent, except
that 256 (for aftertouch) and 257 (for pitch bend) are reserved values.

Data is passed to synthesizer drivers in channel and note descriptors. The following offsets
are used to extract this data.

\ offsets into channel descriptors
15 constant cd-channel \ channel number (8 bits)
16 constant cd-patch \ current physical patch (low 16 bits)
\ and location (high 16 bits)

\ offsets into note descriptors
28 constant nd-pitch \ 16-bit pitch (8 fractional bits)
32 constant nd-volume \ 0 .. 127

The following is a fragment of the generic MIDI synthesizer driver. The complete code is
in sm-drvr0.

create drvr0-buf \ static event record for (MIDI-command)
action-rec-size 24 + allot

code generic-MIDI-driver
drvr0-buf 1# a1 move
nargs-offset a1 d) a0 lea
1 d0 subq
0< wif
3 d0 mov eq
 d0 a0 )+ move
th 90 d0 moveq
 cd-channel a4 d) d0 byte or long
d0 a0 )+ move
 nd-pitch 2 + a3 d) d0 bmove
 th 7f # d0 byte and long
d0 a0 )+ move
 nd-volume a3 d) a0 )+ move
 ' (MIDI-command 1#) jmp
wthen
\ ... handle change-patch and other command codes
\ omitted here for brevity

end-code
8. INPUT HANDLING

FORMULA provides interfaces to mouse, keyboard and MIDI input. There is a single handler process for each type of input. Keyboard input is handled by the FORTH interpreter process. For the other input sources, you can write your own handler process or use a FORMULA default handler. Handlers are created by deferred words called from formula. You can redefine these words to install your own handlers.

A handler process executes an infinite loop, reading and handling input. Whenever there is no input it sleeps, to be later awakened by the corresponding input interrupt routine. If the response to an input event is instantaneous, it can be done by the handler process itself. Otherwise, the input handler should create a separate process to generate the response, thereby freeing itself to handle further input.

8.1. MIDI Input Handling

A MIDI input handler process calls

```
get-MIDI-command ( -- data1 data0 command time | data0 command time ; )
```

to get the next MIDI command (if necessary, the process will sleep until the complete command has been received). The SVT time at which the command arrived is also returned; normally the handler stores this in its time-position.

The deferred word create-MIDI-handler creates the MIDI handler. A MIDI handler can be installed as follows (this must be loaded before running formula):

```
:ap my-MIDI-handler
    :ap
    assign-proc-ID proc-name" MIDI handler"
    immortal
    begin
    get-MIDI-command
    to time-position
    ( ... handle the command )
    again
    ;:ap
    ;:ap
    
    ' my-MIDI-handler is create-MIDI-handler
```

FORMULA's default MIDI handler maintains a table of handler routines, one for each type of MIDI command. When a MIDI command is received, the corresponding word is executed. The following table shows which entry corresponds to which MIDI control byte:

<table>
<thead>
<tr>
<th>index</th>
<th>meaning</th>
<th>data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>keyup</td>
<td>velocity, key-number</td>
</tr>
<tr>
<td>1</td>
<td>keydown</td>
<td>velocity, key-number</td>
</tr>
<tr>
<td>2</td>
<td>polyphonic pressure</td>
<td>pressure, key-number</td>
</tr>
<tr>
<td>3</td>
<td>control change</td>
<td>value, control-number</td>
</tr>
<tr>
<td>4</td>
<td>patch change</td>
<td>patch-number</td>
</tr>
<tr>
<td>5</td>
<td>mono pressure</td>
<td>pressure</td>
</tr>
<tr>
<td>6</td>
<td>pitch bend</td>
<td>bend-lo, bend-hi</td>
</tr>
<tr>
<td>7</td>
<td>system command</td>
<td>none; this command is ignored</td>
</tr>
</tbody>
</table>
The MIDI action word is passed the one or two data bytes associated with the MIDI command each time it is executed. You can write your own MIDI action words and store them in this table. MIDI action words may create note-playing processes, but they themselves should not generate time advances as this will interfere with response to subsequent MIDI commands.

As an example, the following program provides a “MIDI delay line” with user specifiable decay rate, repetition speed, and number of repetitions:

```
quan repno 2 to repno  \ number of times to repeat each note
quan reprate 400 to reprate \ SVT delay between repetitions

:ap rnote \ vel keynumber --- ; repeat note with decay
    ::ap
        [ 2 params ]
        repno 0 do
            2dup mkd \ play the note
            swap \ s: key-no vel
            2 3 */ \ decay velocity a little
            swap \ s: vel key-no
            reprate time-advance \ wait for delay time
            loop
        2drop
    ::ap

\ rnote 1 MIDI-action ! \ install handler for key down
```

FORMULA also provides routines that allow MIDI channel patch commands to automatically reconfigure the MIDI-action table. For each of the system non-system MIDI commands, a table of CFA pointers, indexed by patch number, is maintained. When a command to change to channel-patch \( N \) is received, default-cp examines the \( N \)th entry in each of these arrays and, if it is non-zero, stores it in the appropriate entry of the MIDI-action table. The syntax for assigning an action to a MIDI channel patch number is:

```
n ku-action: <name> \ key up action for patch \( N \n
n kd-action: <name> \ key down action for patch \( N \n
n pp-action: <name> \ polyphonic pressure action for patch \( N \n
n cc-action: <name> \ continuous controller action for patch \( N \n
n cp-action: <name> \ channel patch action when CP \( N \) is received

n at-action: <name> \ aftertouch action for patch \( N \n
n pb-action: <name> \ pitch bend action for patch \( N \n```

For this system to work, every cp-action must call default-cp.

### 8.2. Mouse Input Handling

FORMULA’s mouse interrupt handler is installed by calling +mouse. After this is done, the mouse position and the state of the left and right mouse buttons is maintained in the following quans:

```
mouse-x \ mouse x coordinate
mouse-y \ mouse y coordinate
left-button \ left button down
right-button \ right button down
```

mouse-x and mouse-y can be set to any initial value; mouse motions are handled by making incremental changes to them.

These quans can be “pulled” by any process. You may instead want to have a process that sleeps until there is mouse input. FORMULA provides a default mouse handler process is started
by calling mouse-hnd (which in turn calls +mouse). Whenever a mouse event occurs, this
process calls the deferred word mouse-routine, which you can define to do whatever you
want. In the following example, a pair of notes is played on each mouse event, with pitches
corresponding to the x and y positions of the mouse.

:ap foo
   :ap
       mouse-x 8 / 50 + z$
       mouse-y 8 / 50 +$
   ;:ap
   ;:ap
   foo is mouse-routine

mouse-hnd

8.3. Function Key Handling

The Atari's function keys and shifted function keys can be assigned FORMULA words to
be executed when the key is pressed. Indices from 0 to 9 select function keys 1 to 10 while
indices from 10 to 19 select shifted function keys.

FK-table       ( name ; --- ; create a function key action table )
FK-action:     ( name ; n --- ; insert entry in a function key action table )
default-FK-table ( predefined default function key action table )

FK-table defines a function key action table (initially all entries are noop). Invoking name
activates the table. FK-action: makes name the nth entry of the currently active table.

For example:

default-FK-table
11 FK-action: foo

will activate the default function key table and cause foo to be executed whenever you press the
shift key and function key 2 simultaneously.
9. STILL MORE FEATURES

9.1. Random Number Generators

FORMULA provides the following words for generating random numbers:

```
rnd   ( --- n )
irnd  ( m --- n )
brnd  ( --- flag )
rndinit ( n --- )
grnd  ( m --- n )
frnd2 ( m --- n )
frnd2-init ( --- )
frnd3 ( old n --- new )
trand ( table-addr --- n )
```

*rn*nd returns a 32-bit random integer from the uniform distribution. *ir*nd returns a random integer from the uniform distribution from 0 to *n*-1. *br*nd returns a random 0 or 1 (it's faster than 2 *ir*nd). *rn*ndinit initializes, based on the seed *n*, the table used by *rn*nd, *ir*nd and *br*nd. *gr*nd returns a random number from the Gaussian distribution with mean zero and standard deviation *m*, rounded to the nearest integer. *fr*nd2 returns a number from an integer sequence on (0, ..., *m*-1) with (roughly) 1/2 frequency content. The underlying distribution is uniform. The sequence's "state" is per-process; *fr*nd2-init must be called to initialize this state by each process using *fr*nd2. *fr*nd3 is another 1/f random number generator; its state consists simply of its previous value, which is passed as the *old* argument. *tr*and returns a random number from a distribution on (0, ..., *n*-1) described by a table whose first entry is the sum of weights, and whose remaining *n* entries are weights on each integer in (0, ..., *n*-1).

9.2. Time Control Structures

FORMULA has three *time control structures*:

```
maxtime ( n --- )
...
maxend

mintime ( n --- )
...
imend

mintime ( n --- )
...
imloop
```

*ma*xtime ... *ma*xend specifies that the enclosed code is to consume at most *n* units of time (in the process's units). If this limit is exceeded, control is transferred to the statement following *ma*xend and the stacks are restored to their level at *ma*xtime. For example,

```
: ap foo
  :tg begin 1|4 irnd & again
  8|1 maxtime
    begin c $ again
    maxend
  :ap
```

plays a sequence of random-length notes lasting exactly 8 measures.

*mi*ntime(*n*) ... *mi*nend specifies that the code block is to be extended by a *time-advance*, if necessary, so that it consumes at least *n* units of virtual time.
mintime(n) ... minloop specifies that the code block is to be iterated, if necessary, so that it consumes at least \( n \) time units.

These structures can be nested; outermost structures have priority. Future actions scheduled within a maxtime construct will be performed even if they lie outside the bound. In the example given above, this means that the end times of the notes generated by foo may lie beyond the 8-beat limit.

9.3. Memory Allocation

Most of FORMULA's internal data structures are dynamically allocated. The memory allocation facility may be useful to user programmers as well.

\[
\begin{align*}
\text{malloc} & \quad ( n \quad \text{addr} ; \text{allocate a block of } n \text{ bytes} ) \\
\text{free} & \quad ( \text{addr} \quad \text{---} ; \text{free an allocated block} )
\end{align*}
\]

Memory is allocated from an area that starts at default-free-mem and grows downwards as needed. If FORMULA runs out of memory, it panics (i.e., process scheduling is disabled and interrupt vectors are restored to their original state).
10. EVENT BUFFERING AND PROCESS SCHEDULING

10.1. Event Scheduling

The *computation* of a note (i.e., its pitch, volume etc.) can be distinguished from its *performance* (sending the MIDI commands to start and stop the note). FORMULA exploits this distinction to increase timing accuracy by using a technique called *event buffering*. An *event* is an output action whose computation can be separated from its performance. Processes are allowed to compute events slightly ahead of time. Event descriptors are kept in an *event buffer*, and performed when their time arrives. Each descriptor contains the address of a *performance routine* and a set of parameters to pass to that routine.

The use of event buffering requires that a process wanting to generate an output (e.g., to start a note) does not do it directly, but rather schedules an event to do it. For example, the word `mkd` (§2.1) does not directly generate MIDI output; it schedules an event that will generate the output. Most FORMULA programmers will never have to schedule events directly, since it is done for them by higher-level words such as `mkd` and `$. This section explains the eventscheduling interface, for those who are interested.

Event performance routines may be either high-level Forth, or (for efficiency) machine-code routines. The following constructs are used to schedule Forth events:

```forth
::ev
  ( Forth code for event routine )
;;ev

n ::fev
  ( Forth code for event routine )
;;ev
```

The first construct schedules an event for the current time position of the process; the second schedules an event for n units later. In both cases, `[ n params ]` can be used to pass parameters to the event routine. For example, the following program prints 0 through 9, one number per second:

```forth
:ap
  10 0 do
    i
    ::ev
      [ 1 params ]
    ;;ev
  1000 time-advance
  loop cr
;;ap
```

The following words are used to schedule machine-code events:

- **event** ( args nargs CFA --- ; schedule an event )
- **event-rec** ( args nargs CFA --- rec ; schedule an event )
- **future-event** ( args nargs CFA delay --- ; schedule event after delay )

CFA is the address of a machine-code event performance routine. args is a set of parameters to be passed to that routine. The routine will be called with a pointer to an “event record” in the A1 register; it may modify A0-A2 and D0-D3. An event descriptor has several fields; the ones relevant to event performance routines are the number of arguments and the argument block. The

---

20 Event routines are executed at interrupt level and they cannot call TOS routines (e.g., terminal and disk I/O).
offsets of these fields are stored in the constants nargs-offset and args-offset respectively. The order of arguments is the same as their stack order. The word must return with an rts instruction. In addition, it normally must free the event record using the (free) macro, which expects the record pointer to still be in A1.

event-rec is like event but returns a pointer to the event record\(^{21}\). future-event is like event, but causes the event to be scheduled delay units in the future.

As an example, mkd is defined as

\begin{verbatim}
:ap mkd
  th 90 or 3 ' MIDI-command event
:ap
\end{verbatim}

where MIDI-command is a machine-code routine, written according to the above rules, that outputs a MIDI command.

10.2. Synchronization

Processes may be interrupted by I/O and clock interrupts. In addition, a process may preempt another process. Hence any data structures that are shared between a process and an interrupt handler, or between two processes, need a synchronization mechanism. FORMULA provides several synchronization mechanisms.

10.2.1. Mutual Exclusion Between Processes

Mutual exclusion is needed for “critical sections” of code in which at most one process can be allowed to execute at once. Since process preemption is done using a “software interrupt” mechanism (§11.1.3), mutual exclusion between processes can be achieved by masking software interrupts:

\begin{verbatim}
mask-softint ( --- ; raise software interrupt mask level )
unmask-softint ( --- ; lower software interrupt mask level )
\end{verbatim}

These must always occur in matched pairs, which can be nested. This method should be used only for fairly short critical sections.

For potentially time-consuming critical sections, real-time semaphores should be used. A process holding a semaphore can be preempted (in contrast with software interrupt masking, which disables preemption). If a process A tries to acquire a semaphore held by a preempted process B, B is temporarily “promoted” to the deadline of A, allowing it to run until it releases the semaphore. The operations are:

\begin{verbatim}
semaphore: ( name ; --- ; declare a semaphore )
init-sema ( semaphore-addr --- ; initialize a semaphore )
P ( semaphore-addr --- ; acquire the lock )
V ( semaphore-addr --- ; release the lock )
\end{verbatim}

semaphore: creates a named semaphore. The resulting word, when invoked, returns the semaphore address. The semaphore must be initialized (at runtime) before it is used.

10.2.2. Synchronizing with Input Sources

The above mechanisms cannot be used to synchronize with interrupt handlers. The following words provide a synchronization mechanism for interrupt-driven input with a single handler process (these are used in the FORMULA implementation of keyboard, MIDI and mouse input,

\(^{21}\) This is used by the SM, for which the argument to the note-end event is a pointer to the event record for the note-start event.
and can be used for other I/O devices):

\begin{verbatim}
input-handler ( name ; --- ; declare an input handler structure )
wait-for-input ( handler-structure --- ; )
(wake-input-handler) ( assembly language macro )
\end{verbatim}

input-handler allocates a handler structure that maintains a pointer to the handler process and a counter for the amount of data available. wait-for-input is called from the handler process; if no input is available, the process sleeps. (wake-input-handler) is used in the input interrupt handler whenever new input has been received. It wakes up the handler process if necessary.

10.2.3. Interrupt Masking

The synchronization mechanism of last resort is to mask hardware interrupts:

\begin{verbatim}
set-mask ( --- old-mask ; mask interrupts )
(set-mask) ( assembly language macro version )
restore-mask ( old-mask --- ; restore old mask )
(restore-mask) ( assembly language macro version )
\end{verbatim}

Interrupts (and therefore preemptions) cannot occur between set-mask and restore-mask. Interrupt-masking should be used only for short critical sections (100 instructions or less) because of the possibility of lost interrupts, especially from MIDI input.

10.3. Process Scheduling Parameters

Each top-level process has several "scheduling parameters", stored in the following pquans (all times are in SVT units):

- \textit{time-position} is the time for which the process is currently computing events.
- \textit{deadline} determines the order of process execution: the executing process is that with the earliest deadline (§11.3.2).
- \textit{maxdel} determines how far ahead of the current SVT the process is allowed to compute. If the process advances to a time position such that

\[ \text{time-position} > (\text{current SVT}) + \text{maxdel} \]

then the process is suspended, and remains \textit{dormant} until the condition no longer holds. \text{maxdel} must be nonnegative.

- \textit{mindel} determines the process's deadline as a function of its time position:

\[ \text{deadline} = \text{time-position} - \text{mindel} \]

A process's \textit{mindel} must be strictly less than its \textit{maxdel}, and can be negative.

A quan \textit{system-mindel} stores the maximum amount by which processes are allowed to fall "behind schedule" before the advance of system virtual time is stopped to allow them to catch up. Specifically, SVT is not advanced if

\[ \text{earliest deadline} + \text{system-mindel} < \text{current SVT} \]

The \textit{maxdel} and \textit{mindel} parameters of top-level processes can be adjusted to obtain a "scheduling policy" that is optimized for your particular application. (A process belonging to a group is governed by the parameters of its top-level ancestor group.)

- The relative values of \textit{mindel} can be used to "prioritize" input-handling processes. Suppose response to MIDI input is more time-critical than response to mouse input. By giving the MIDI input handler process a larger (more positive) \textit{mindel} than the mouse handler, the MIDI handler will have an earlier deadline (and will therefore run first) whenever MIDI and mouse events occur at about the same time.
The value of maxdel is a tradeoff between timing accuracy and response latency. If maxdel is small, the process will respond quickly to input but may experience event timing errors if system load is heavy or its own computations are long. If maxdel is large, the process will be more immune to timing errors, at the expense of increased input response time.

By changing system-minidel you can determine what happens when the system falls behind schedule. If system-minidel is nonnegative, then SVT stops advancing whenever the system falls behind schedule (i.e., when SVT exceeds the earliest process deadline). All subsequent events are uniformly shifted in time, but the system loses synchronization with external timing sources. If you need such synchronization, system-minidel should be given a large negative value. This may cause short-term compression of events if lateness occurs.

maxdel and minidel can be changed at any time. For example, the maxdel of a process can be temporarily reduced during a period of frequent user interaction with the process. A process must call

```
0 time-advance
```

to have these changes take effect (that is, to change its deadline and perhaps become dormant).

10.4. Background Processes

FORMULA’s deadline scheduling mechanism is designed for processes that schedule output events for specific times, using relatively small amounts of CPU time to compute the events. Unfortunately, not all processes conform to this model. For example, the Forth interpreter process, while it is executing words (which prints the contents of a vocabulary) uses large amounts of CPU time and has no particular output timing requirements.

Such processes can be accommodated by making them into background processes:

```
background  ( --- ; make caller into a background process )
foreground  ( --- ; make caller into a foreground process )
```

The deadline of a background process is always "infinite"; the range 0x7fffffff00 to 0x7fffffffff is reserved for background processes. The time position of a background process is initially less than current SVT. When a background process does something of a real-time nature (scheduling an event, calling time-advance, or creating a group) its time position is set to the maximum of its current value and the current SVT. This arrangement allows background processes to generate real-time output; e.g., you can play music in the Forth interpreter. The computation of the music, however, is done with an infinite deadline so it may potentially be "starved" by other processes.
11. FORMULA IMPLEMENTATION

11.1. Scheduling and Event Performance

This section briefly describes FORMULA's implementation of process and event scheduling. Three "global" time systems are used:

- **Real time** (or "wall clock" time) is maintained by a 200 Hz periodic clock interrupt.
- **System time** (ST) normally advances at the same rate as real time. However, when a process has fallen behind schedule (that is, when the current ST exceeds the "deadline" of the process), ST stops advancing to allow the process to catch up.
- **System virtual time** (SVT) advances with ST, but with a scaling factor **tempo** (units of SVT per unit of ST). **tempo** provides a high-resolution global tempo control. It can be set directly (it is a quan) or using **usecs-per-SVT** ($\S2.1$).

11.1.1. Data Structures

FORMULA uses the following record types. The pquan mechanism ($\S3.4$) is used to access the fields of these records.

- **Context blocks**: each stores the state of a process, including its stacks and pquans.
- **Event records**: each describes a pending event, including its parameters and a pointer to its performance routine.

These records are organized in the following structures:

- The **execution queue** is a list of context blocks of executable processes, sorted by increasing value of deadline. The executing process (the **execution queue head**) is always that with the earliest deadline.
- The **wakeup buffer** stores event records and context blocks of dormant processes. It is implemented as an array of lists of entries. Each entry has a **wakeup-time** field storing the SVT when the event is to be performed or the process is to be made executable. The array position of an entry is given by the low order bits of its wakeup time.

11.1.2. Time-Advance and Event Scheduling

FORMULA's time-related primitives are implemented as follows. First, **time-advance** stores its argument in the **delay** pquan, applies the caller's TD's to **delay** and adds the result to its **time-position**. If the caller is in a group, the delay until the new earliest member of the group is propagated to the group's context block; this recurses up the tree. At the top level, the assignments

\[
\text{deadline} = \text{time-position} - \text{mindel}; \\
\text{wakeup-time} = \text{time-position} - \text{maxdel};
\]

are done. If **wakeup-time** is greater than the current SVT, the object is moved to the wakeup buffer; otherwise the object is reinserted in the execution queue. In either case, a context switch is done to the new execution queue head.

**event** allocates an event record and moves its arguments to the record. The wakeup time of the event record is set to the maximum of the caller's time position and the current SVT. The event record is then inserted in the wakeup buffer.

11.1.3. Event Performance and Preemption

The FORMULA scheduler executes processes in order of increasing deadline, and a process is preempted if one with an earlier deadline becomes runnable. Action performance and process
preemption are done jointly by the clock interrupt handler and the software interrupt handler. On each clock interrupt, the handler sees if the current SVT is less than the earliest process deadline. If so, it adds tempo to the current SVT, and checks if any entries in the wakeup buffer array for times between the previous and current SVT's are non-empty, and requests a software interrupt if so.

The software interrupt handler processes lists of records from the wakeup buffer. It moves dormant processes to the execution queue and executes event performance routines. The software interrupt routine continues to process lists from successive entries of the wakeup buffer until its index exceeds the current SVT. If, on completion of the software interrupt handler, there is a new execution queue head, then the old execution queue head is preempted and a context switch is done.

When a process is awakened by an interrupt routine, it is moved to the wakeup buffer rather than directly to the execution queue. Its wakeup time is assigned so that it will be awakened on the next clock tick.

`kill` and `suspend` set flags in the context block. These flags are checked, and the appropriate action taken, the next time the object is removed from the wakeup buffer.

11.2. The Accuracy and Range of Time Specification

The timing accuracy of events is limited by the 5 millisecond resolution of the Atari's interrupt clock. Forthmacs uses 32-bit integers, producing the following limitations:

- Time positions are integers, so SVT wraps around (producing indeterminate results) every \(2^{32}\) SVT units, roughly 1000 hours if SVT unit = 1 ms.
- In rational conversion by \(r>i\), the term numerator\(\times rscale\) must fit in 32 bits, and the denominator must fit in 16 bits.
- Time deformations use 8-bit fixed-point multipliers. Therefore time intervals passed through TD's must fit in about 23 bits (depending on the magnitude of the tempo functions).

The following table shows representative time interval representations for \(rscale = 2000\) and SVT unit = 1000 microseconds (the defaults).

<table>
<thead>
<tr>
<th></th>
<th>whole note</th>
<th>64th note</th>
</tr>
</thead>
<tbody>
<tr>
<td>rational</td>
<td>111</td>
<td>164</td>
</tr>
<tr>
<td>SVT</td>
<td>2000</td>
<td>-31</td>
</tr>
<tr>
<td>real-time</td>
<td>2 sec.</td>
<td>-.03 sec.</td>
</tr>
</tbody>
</table>

With these parameters, the largest usable interval is about 4000 whole notes if time deformations are used (because of the 23-bit limitation).

There are several sources of round-off error in FORMULA:

- The round-off accumulated by \(r>i\) is 16 bits, leaving uncompensated error up to \(2^{-16}\) of an SVT unit.
- TD's accumulate only 8 bits of error. This is a problem only when multiple processes use different TD's that should synchronize periodically.

\[\text{22 A software interrupt is requested in software, and has lower priority than any hardware interrupt. On the Atari ST, this is done using the horizontal retrace interrupt, which is normally not used, has lower priority than any other interrupt, and is triggered at a very high rate.}\]
APPENDIX A: DISTRIBUTION AND COPYING

The current release of FORMULA is available from the authors. The copying policies for FORMULA and Forthmacs are described in the file `copying.txt`. Both packages are copyrighted, but the respective authors have granted permission for the release disks to be freely used, copied, and given away. However, they may not be sold.
APPENDIX B: FORTH AND FORTHMACS

1. FORTH

FORMULA is an extension of the programming language FORTH. You can use FORMULA without knowing FORTH, but you will be somewhat limited. A good introductory book on FORTH is Mastering FORTH by Anita Anderson and Martin Tracy, published by Prentice Hall.

FORTH is a very simple programming language. FORTH's program unit is the word. Words communicate using a data stack; most words take their arguments from the stack and leave their result on the stack. Return addresses are kept on a separate return stack.

The documentation of a word often includes a 1-line summary:

name ( old --- new ; comments )

Old and new represent the stack contents before and after executing the word. For example, the summary of the words + and over are:

+ ( n m --- sum ; replace top 2 stack elements by their sum )
over ( n m --- n m n ; duplicate 2nd-to-top stack element )

Some words take a text argument (usually the name of a word or a file) that follows them in the command or source file. For example, the word array is used as follows:

10 array foo ( declare an array of 10 words )

The 1-line summary of array is:

array ( name ; nwords --- ; declare an array )

FORTH systems are interactive, and include an interpreter that processes your keyboard input. The interpreter lets you invoke a word simply by typing its name. It also lets you enter word definitions via the keyboard. Most FORTH systems also include built-in words that let you create and edit disk files, and load word definitions from disk files.

2. FORTHMACS

FORTHMACS is an implementation of the FORTH-83 standard with many enhancements, and includes a version of the Emacs text editor. FORTHMACS has many features that set it apart from other FORTH implementations:

- File system interface

FORTHMACS uses named files instead of disk blocks, and provides the following file commands:

load ( file-name ; --- ; load (interpret) a file )
ed ( --- ; invoke the Emacs text editor )
more ( file-name ; --- ; view a file w/ pagination )
cd ( directory-name ; --- ; enter a subdirectory )
rm ( file-name ; --- ; delete a file )
mv ( oldname newname ; --- ; rename a file )
lst ( --- ; list directory contents )

You can use absolute and relative pathnames as in UNIX, except that \ rather than / separates components.

This section is to help you get started with FORTHMACS; you should buy the complete documentation if you plan on using FORMULA a lot.

Emacs documentation is included in the FORTHMACS documentation.
• Command completion
If you type the start of a word name, <control><space> will extend it to a complete name if possible. This saves typing when using long names. <control>? prints the names that extend the current word fragment.

• Command history
<control>p and <control>n cycle back and forth through a list of recent command lines; this can be used to avoid retyping long commands.

• Command-line editing
Command lines (including those resurrected by <control>p) may be edited with intra-line Emacs commands; <return> causes the entire line to be executed, regardless of cursor position.

• Decompilation and On-Line Documentation
The following words facilitate on-line source code browsing:

```forth
see   ( name ; --- ; decompile a word )
calls ( CFA --- ; )
whatis ( name ; --- ; print 1-line summary )
```

See can, in most cases, completely reconstruct a word’s definition from its compiled form. This allows you to conveniently peruse the source code of Forthmacs and FORMULA. .calls lists all words that call a given word. whatis prints a 1-line summary of a word (this uses the files whatis.doc and whatis.ind on the release disk; it does not work for FORMULA words).

• Deferred Words
Forthmacs allows a form of forward reference using deferred words. Such a word is declared by

```forth
defer foo
```

It may then be used in other word definitions. It must later be bound to an actual definition (say blah) using

```forth
' blah is foo
```

• Structured 68000 Assembler
Forthmacs provides a 68000 assembler that uses Forth-like control structures for generating conditional branches. The assembler uses 8-bit branch offsets. This is inadequate for control structures in large words, so FORMULA provides a set of structures using 16-bit offsets; see the file lbranch.
APPENDIX C: THE MIDI STANDARD

MIDI (Musical Instrument Digital Interface) is an industry standard for connecting devices such as synthesizers, keyboards and computers. Connecting the MIDI out socket of one device to the MIDI in socket of another establishes a MIDI connection. Additional instruments can be "daisy-chained" by connecting the MIDI thru socket of one to the MIDI in socket of the next.

Information is transmitted between devices in the form of MIDI commands. Each command is a sequence of 8-bit bytes. MIDI provides 16 distinct channels that allow commands to be directed to specific synthesizers in a chain. Commands consist of an ID for the type of command being delivered, a MIDI channel number, and whatever parameters are required to describe the command. For example, a MIDI "key down" command contains the command ID, the MIDI channel number, the pitch to be played, and the "velocity" with which it should be played (this normally determines the volume of the note).

Each MIDI command is encoded by a command byte (always with the high-order bit set, and therefore in the range 128-255) followed by some data bytes (always in the range 0-127). The following table shows the format of standard MIDI commands. The low-order 4 bits of a command byte are the number of the channel to which the command applies.

<table>
<thead>
<tr>
<th>command byte (x = channel number)</th>
<th>meaning</th>
<th>data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x</td>
<td>keyup</td>
<td>velocity, key-number</td>
</tr>
<tr>
<td>9x</td>
<td>keydown</td>
<td>velocity, key-number</td>
</tr>
<tr>
<td>ax</td>
<td>polyphonic pressure</td>
<td>pressure, key-number</td>
</tr>
<tr>
<td>bx</td>
<td>control change</td>
<td>value, control-number</td>
</tr>
<tr>
<td>cx</td>
<td>patch change</td>
<td>patch-number</td>
</tr>
<tr>
<td>dx</td>
<td>mono pressure</td>
<td>pressure</td>
</tr>
<tr>
<td>ex</td>
<td>pitch bend</td>
<td>bend-lo, bend-hi</td>
</tr>
<tr>
<td>fx</td>
<td>system command</td>
<td>varies</td>
</tr>
</tbody>
</table>

In the MIDI numbering scheme, 60 is middle C.
APPENDIX D: FORMULA SOURCE FILES

The directory formula.dir contains the following files:

patches  bug fixes for Forthmacs
features  words of general (non-FORMULA-specific) utility
lbranch  assembler words to use 16-bit branch offsets
while   multi-exit 'while' statements
pquan   quans and pquans
decls   declarations of deferred and aliased words
interrupt  interrupt masking, etc.
panic   handler for undo key, out-of-memory panic, etc.
malloc   memory allocation
flags   declarations of compile-time flags
proc-cb  basic context block format
stack   words to copy between stacks and memory blocks
process  context-switching primitives
queue   manipulation of various types of queues
aux     definitions of auxiliary-process pquans
tempo  SVT-related stuff
execute  execution queue
wakeup  wakeup buffer
schedint  scheduling interrupt handlers (clock, software interrupt)
rt-sched  basic process scheduling
background  background processes
event   event scheduling and ::ev
gp-sched  group scheduling
gp-creat  group creation
procuser  process/group naming, ID's, pquan manipulation, listing
jobcntrl  suspend, resume, kill
define  ::ap, ::ap and params
faq     future action queue
fevent  ::fev
timebnd  time control structures
handler  synchronization primitives for input handlers
midi-in  low-level MIDI input
midiout1  MIDI output driver
midiout2  interfaces to MIDI output driver (events, macros etc.)
6850-int  6850 interrupt handler
midi-hnd  high-level MIDI input
semaphor  real-time semaphores
tty-out  terminal output with semaphore protection
tty-in   terminal input handling
func-key  function key handling
airshaft  crude windowing facility
mouse   mouse handling
rational  rational to SVT conversion
fraction  A\B, A\B, A.B notation
ptchname  symbolic pitch names
slots    auxiliary process stuff
<table>
<thead>
<tr>
<th>sg</th>
<th>sequence generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>sg-set</td>
<td>install sequence generators</td>
</tr>
<tr>
<td>shape</td>
<td>shapes</td>
</tr>
<tr>
<td>sh-set</td>
<td>install shapes</td>
</tr>
<tr>
<td>td</td>
<td>time deformations</td>
</tr>
<tr>
<td>td-set</td>
<td>install time deformations</td>
</tr>
<tr>
<td>dlr-quan</td>
<td>quans and pquans used by $\text{-words}$</td>
</tr>
<tr>
<td>dollar</td>
<td>the $\text{-words}$</td>
</tr>
<tr>
<td>dlr-dmo</td>
<td>interface of $\text{-words}$ to DMO</td>
</tr>
<tr>
<td>notation</td>
<td>more $\text{-words}$</td>
</tr>
<tr>
<td>sm-data</td>
<td>SM data structures and configuration words</td>
</tr>
<tr>
<td>sm-drvr0</td>
<td>generic MIDI synthesizer driver</td>
</tr>
<tr>
<td>sm-event</td>
<td>SM event routines</td>
</tr>
<tr>
<td>dlr-sm</td>
<td>interface of $\text{-words}$ to SM and DSM</td>
</tr>
<tr>
<td>generic</td>
<td>default SM initialization words</td>
</tr>
<tr>
<td>tuning</td>
<td>tuning systems</td>
</tr>
<tr>
<td>scales</td>
<td>definitions of some tuning systems</td>
</tr>
<tr>
<td>pitchset</td>
<td>pitch-set words</td>
</tr>
<tr>
<td>debug</td>
<td>debugging words</td>
</tr>
<tr>
<td>decomp</td>
<td>decompiler (&quot;see&quot;) extensions</td>
</tr>
<tr>
<td>random</td>
<td>random number generators</td>
</tr>
<tr>
<td>formula</td>
<td>system initialization</td>
</tr>
<tr>
<td>init-def</td>
<td>context block initialization</td>
</tr>
</tbody>
</table>
APPENDIX E: DEBUGGING FORMULA PROGRAMS

1. Crash Analysis

Debugging FORMULA programs can be difficult because of multiple processes, asynchrony, and preemption. With experience, it's not much harder than debugging single-process programs. The following is a brief list of suggestions.

- Many bugs result in stack overflow or underflow. This can often be detected using .cb (§3.3), or by printing the stack pointer values (returned by sp@ and rp@) within a process.

- If the computer “goes dead” (i.e., stops responding to input) but the cursor is still blinking, the likely causes are 1) an infinite loop in a process or an event routine, and 2) a mask-softint without a corresponding unmask-softint. The undo key should work in this case.

- If the undo key fails, the likely causes are 1) an infinite loop in an interrupt handler or in a section of code that masks interrupts, and 2) a set-mask without a corresponding restore-mask. In this case all you can do is reboot.

- The quan who points to the currently executing process, and the quan execution-queue-head points to the top-level object containing this process (perhaps the process itself). It is often helpful to examine these context blocks after an exception.

Many bugs result in hardware exceptions (bus error, illegal instruction). Forthmacs handles these exceptions and saves the machine state, allowing it to be examined later (even after a reboot\(^\text{25}\)) using

```
showcrash ( --- ; print hex dump of 68000 state at last exception )
```

This shows the values of the 68000 registers and stacks at the time of the exception. The registers of greatest interest are the program counter (PC), and Forth's “virtual PC” (the A5 register). These PC points into the code word being executed at time of the crash, and A5 points into the high-level Forth word from which it was called. The easiest way to figure out the words involved is to use

```
dump ( start-adr nbytes --- ; print hex/ASCII memory dump )
```

to survey the relevant areas of memory, looking for dictionary headers (which contain the ASCII word names). You'll want to use hex to switch to hexadecimal mode. Once you've found out exactly where your program was when it crashed, and you've examined the contents of the registers and stacks at that point, it's usually easy to figure out what went wrong.

During debugging it is helpful to know the internals of Forthmacs (i.e., the structure of its dictionary and its register usage). This information is in the Forthmacs manual (see Appendix A).

2. Recompiling FORMULA

The FORMULA source is divided into many files (see Appendix D). Each file foo begins by creating a symbol _foo in the loaded vocabulary. Typing

```
recompile _foo
```

will forget back to that point in the dictionary, and will recompile that part of FORMULA. It also calls restore, so you must run formula again after recompiling.

\(^{25}\) Stack dumps may be meaningless after a reboot, because memory has been zeroed.
You can generate stripped-down versions of FORMULA by editing the file flags. This file contains compile flags whose presence enables the compilation of a particular feature. For example, the $-words are compiled only if _DOLLAR_ is defined.

Compile errors may leave the Forth interpreter with an undesired vocabulary search list (§3.6). You can examine the search list using order. You can restore it to the standard state by typing

```
only forth also definitions
```
APPENDIX F: FORMULA GLOSSARY

The following is a list of the words defined by FORMULA. Each word name is followed by the name of the source file in which it is defined.

<table>
<thead>
<tr>
<th>Word</th>
<th>Description</th>
<th>Source File</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>dollar</td>
<td>scales</td>
</tr>
<tr>
<td>$</td>
<td>dollar</td>
<td>scales</td>
</tr>
<tr>
<td>$*k</td>
<td>dollar</td>
<td>scales</td>
</tr>
<tr>
<td>$2</td>
<td>notation</td>
<td>slots</td>
</tr>
<tr>
<td>$5</td>
<td>notation</td>
<td>procuser</td>
</tr>
<tr>
<td>$8</td>
<td>notation</td>
<td>procuser</td>
</tr>
<tr>
<td>$DMO</td>
<td>dlr-dmo</td>
<td>patches</td>
</tr>
<tr>
<td>$DSM</td>
<td>dlr-sm</td>
<td>patches</td>
</tr>
<tr>
<td>$SM</td>
<td>dlr-sm</td>
<td>procuser</td>
</tr>
<tr>
<td>$channel</td>
<td>dlr-quan</td>
<td>define</td>
</tr>
<tr>
<td>$cvolume</td>
<td>dlr-quan</td>
<td>define</td>
</tr>
<tr>
<td>$transpose</td>
<td>dlr-quan</td>
<td>event</td>
</tr>
<tr>
<td>$location</td>
<td>dlr-quan</td>
<td>fevent</td>
</tr>
<tr>
<td>$n</td>
<td>dlr-quan</td>
<td>define</td>
</tr>
<tr>
<td>$n*k</td>
<td>dlr-quan</td>
<td>sh-set</td>
</tr>
<tr>
<td>$note-routine</td>
<td>dlr-quan</td>
<td>td-set</td>
</tr>
<tr>
<td>$nroll</td>
<td>dlr-quan</td>
<td>decomp</td>
</tr>
<tr>
<td>$patch</td>
<td>dlr-quan</td>
<td>sg-set</td>
</tr>
<tr>
<td>$pedal</td>
<td>dlr-quan</td>
<td>generic</td>
</tr>
<tr>
<td>$pedal-routine</td>
<td>dlr-quan</td>
<td>generic</td>
</tr>
<tr>
<td>$pitch-convert</td>
<td>dlr-sm</td>
<td>midout1</td>
</tr>
<tr>
<td>$priority</td>
<td>dlr-quan</td>
<td>midout1</td>
</tr>
<tr>
<td>$transpose</td>
<td>dlr-quan</td>
<td>midout1</td>
</tr>
<tr>
<td>$volume</td>
<td>dlr-quan</td>
<td>semaphor</td>
</tr>
<tr>
<td>&amp;</td>
<td>sg</td>
<td>rt-sched</td>
</tr>
<tr>
<td>(#keys-in-buffer)</td>
<td>tty-in</td>
<td>generic</td>
</tr>
<tr>
<td>(#newkeys-in-buffer)</td>
<td>tty-in</td>
<td>generic</td>
</tr>
<tr>
<td>($n</td>
<td>dollar</td>
<td>semaphor</td>
</tr>
<tr>
<td>(..gp*)</td>
<td>procuser</td>
<td>patches</td>
</tr>
<tr>
<td>(:::sh</td>
<td>sh-set</td>
<td>patches</td>
</tr>
<tr>
<td>(:::td</td>
<td>sh-set</td>
<td>patches</td>
</tr>
<tr>
<td>(:::sh</td>
<td>bake</td>
<td>patches</td>
</tr>
<tr>
<td>(:::td</td>
<td>add-empty-multi</td>
<td>patches</td>
</tr>
<tr>
<td>[]</td>
<td>add-single</td>
<td>patches</td>
</tr>
<tr>
<td>[]</td>
<td>(adr</td>
<td>sm-event</td>
</tr>
<tr>
<td>[]</td>
<td>formula</td>
<td>sm-event</td>
</tr>
<tr>
<td>[]</td>
<td>(arouse-process)</td>
<td>sm-event</td>
</tr>
<tr>
<td>[]</td>
<td>(ascmin</td>
<td>pquan</td>
</tr>
<tr>
<td>(byye</td>
<td>patches</td>
<td>wakeup</td>
</tr>
<tr>
<td>((cr</td>
<td>patches</td>
<td>pitchset</td>
</tr>
<tr>
<td>((cursor-off</td>
<td>patches</td>
<td>procuser</td>
</tr>
<tr>
<td>((cursor-on)</td>
<td>patches</td>
<td>patches</td>
</tr>
<tr>
<td>(esc</td>
<td>patches</td>
<td>pitchset</td>
</tr>
<tr>
<td>(init-active</td>
<td>patches</td>
<td>slots</td>
</tr>
<tr>
<td>(just</td>
<td>scales</td>
<td>patches</td>
</tr>
<tr>
<td>(just1</td>
<td>scales</td>
<td>pitchset</td>
</tr>
<tr>
<td>(if</td>
<td>scales</td>
<td>sm-event</td>
</tr>
<tr>
<td>(ipelog-barang</td>
<td>scales</td>
<td>patches</td>
</tr>
<tr>
<td>(pent</td>
<td>scales</td>
<td>patches</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>-e-</td>
<td>ptchname</td>
<td>/2</td>
</tr>
<tr>
<td>-f-</td>
<td>ptchname</td>
<td>/2+</td>
</tr>
<tr>
<td>-fraction</td>
<td>fraction</td>
<td>/2+</td>
</tr>
<tr>
<td>-g-</td>
<td>ptchname</td>
<td>/2,..</td>
</tr>
<tr>
<td>-mouse</td>
<td>mouse</td>
<td>/2.</td>
</tr>
<tr>
<td>-n</td>
<td>ptchname</td>
<td>/2.4</td>
</tr>
<tr>
<td>-nps</td>
<td>pitchset</td>
<td>/32</td>
</tr>
<tr>
<td>-oct</td>
<td>ptchname</td>
<td>/4</td>
</tr>
<tr>
<td>-perf</td>
<td>interrupt</td>
<td>/4+</td>
</tr>
<tr>
<td>-ps</td>
<td>pitchset</td>
<td>/4-3</td>
</tr>
<tr>
<td>-semaphore</td>
<td>semaphor</td>
<td>/4-3</td>
</tr>
<tr>
<td>-shaft</td>
<td>airshaft</td>
<td>/4..</td>
</tr>
<tr>
<td>-tty</td>
<td>tty-in</td>
<td>/4.8</td>
</tr>
<tr>
<td>.:ap</td>
<td>decomp</td>
<td>/4.8</td>
</tr>
<tr>
<td>.:ev</td>
<td>decomp</td>
<td>/64</td>
</tr>
<tr>
<td>.:gp</td>
<td>decomp</td>
<td>/8</td>
</tr>
<tr>
<td>.:sh</td>
<td>decomp</td>
<td>/8</td>
</tr>
<tr>
<td>.:sh</td>
<td>decomp</td>
<td>/8+</td>
</tr>
<tr>
<td>.:td</td>
<td>decomp</td>
<td>/8-3</td>
</tr>
<tr>
<td>.:td</td>
<td>decomp</td>
<td>/8-3</td>
</tr>
<tr>
<td>.:tag</td>
<td>decomp</td>
<td>10/16</td>
</tr>
<tr>
<td>.:tag</td>
<td>decomp</td>
<td>11$</td>
</tr>
<tr>
<td>.:ap</td>
<td>decomp</td>
<td>12/16</td>
</tr>
<tr>
<td>.:ev</td>
<td>decomp</td>
<td>16/32</td>
</tr>
<tr>
<td>.:gp</td>
<td>decomp</td>
<td>16/8</td>
</tr>
<tr>
<td>.:sh</td>
<td>decomp</td>
<td>2$</td>
</tr>
<tr>
<td>.:td</td>
<td>decomp</td>
<td>2/16</td>
</tr>
<tr>
<td>.:tag</td>
<td>decomp</td>
<td>2/2</td>
</tr>
<tr>
<td>.adr</td>
<td>decomp</td>
<td>2/32</td>
</tr>
<tr>
<td>.all</td>
<td>procuser</td>
<td>2/4</td>
</tr>
<tr>
<td>.aux</td>
<td>slots</td>
<td>2/8</td>
</tr>
<tr>
<td>.cb</td>
<td>procuser</td>
<td>32/16</td>
</tr>
<tr>
<td>.eq</td>
<td>debug</td>
<td>4/32</td>
</tr>
<tr>
<td>.gp</td>
<td>procuser</td>
<td>4/4</td>
</tr>
<tr>
<td>.iadr</td>
<td>decomp</td>
<td>5$</td>
</tr>
<tr>
<td>.ito</td>
<td>decomp</td>
<td>5/8</td>
</tr>
<tr>
<td>.padr</td>
<td>decomp</td>
<td>6/16</td>
</tr>
<tr>
<td>.pget</td>
<td>decomp</td>
<td>6850-handler</td>
</tr>
<tr>
<td>.poffset</td>
<td></td>
<td>7/8</td>
</tr>
<tr>
<td>.pquan</td>
<td>decomp</td>
<td>8$</td>
</tr>
<tr>
<td>.ps</td>
<td>pitchset</td>
<td>8/16</td>
</tr>
<tr>
<td>.pto</td>
<td>decomp</td>
<td>8&lt;&lt;</td>
</tr>
<tr>
<td>.quan</td>
<td>decomp</td>
<td>8&gt;&gt;</td>
</tr>
<tr>
<td>.stack</td>
<td>procuser</td>
<td>:-(</td>
</tr>
<tr>
<td>.to</td>
<td>decomp</td>
<td>::ap</td>
</tr>
<tr>
<td>.wq</td>
<td>debug</td>
<td>::ap-init-active</td>
</tr>
<tr>
<td>/1</td>
<td>notation</td>
<td>::ash</td>
</tr>
<tr>
<td>/1</td>
<td>notation</td>
<td>::ev</td>
</tr>
<tr>
<td>/1,..</td>
<td>notation</td>
<td>::fev</td>
</tr>
<tr>
<td>/16</td>
<td>notation</td>
<td>::gp</td>
</tr>
<tr>
<td>/16.</td>
<td>notation</td>
<td>::gp-init-active</td>
</tr>
</tbody>
</table>
sh-set P
sh-set PV#line
td-set PV#out
td-set PV(emit)
sh-set PV(esc)
sh-set PVat
sh-set Pemit
td-set RT-active-exit
td-set RT-awaken
sg-set RT-sleep
define RT-time-advance
sg SM-$note-routine
shape SM-$pedal-routine
td SM-note-end
define SM-note-prep
event SM-note-start
define SM-param-change
sg-set SM-pedal-down
sh-set SM-pedal-up
td-set V
shape Vemit
sg a-
td absolute
shape active-FK-table
bgnd active-create
active-exit
dlr-dmo addr
dlr-dmo dlr-dmo
dlr-dmo dlr-dmo
dlr-sm dlr-sm
dlr-sm
sm-event
sm-event
sm-event
sm-event
sm-event
sm-event
sm-event
sm-event
sm-event
func-key
func-key
func-key
beats-per-minute
bitmask
bitprob
block-key?
blues
branch-addr
brnd
c$s
c-
carray
cbsize
s
tsemaphor
airshaft
airshaft
airshaft
tty-out
tty-out
rt-sched
rt-sched
rt-sched
rt-sched
rt-sched
dlr-sm
dlr-sm
dlr-sm
dlr-sm
ptchname
shape
func-key
rt-sched
decls
pquan
pquan
pitchset
proc-cb
proc-cb
proc-cb
proc-cb
features
sh-set
slots
procuser
decls
ptchname
backgrnd
rational
random
random
tty-in
pitchset
timebnd
random
dollar
ptchname
features
pquan
wrepeat     while
wthen       lbranch
wuntil      lbranch
wwhile      lbranch
wwhile      while
dollar
APPENDIX G: DIFFERENCES BETWEEN VERSION 3.4 AND PREVIOUS VERSIONS

If you have written programs under an earlier version of FORMULA you will have to change them somewhat for version 3.4. The following are some of the major differences:

1) All process creation is done using :: constructs. Executing a :ap word from the interpreter no longer causes it to run as a separate process. [ n params ] is relevant only within a :: construct.

2) The sh1 foo notation for installing auxiliary processes no longer exists. You must use ::sh1, etc.

3) A TSG word cannot be directly called within an active process definitions. It can be called only within ::tsg ... ::sg or :sg ... :sg.

4) create-group no longer exists. Use ::gp.

5) Timing sequence generators no longer have an implicit infinite loop around them. If a TSG reach the end of its definition, it exits and quarter-note durations are used.

6) A volume shape or TD disappears (and ceases to have an effect) when the end of its definition is reached.

7) Symbolic pitch names no long refer to the “closest instance” of the named pitch; they now refer to the instance within the current octave.

8) Instead of directly assigning to rscale and tempo, you should use beats-per-minute and usecs-per-SVT.

9) Words like d/8 no longer exist. Use 1\|8 instead.

10) Use 1\|8 instead of 1 8 r>1.

11) Many words relating to $ have been renamed; for example, volume becomes $volume, patch becomes $patch, etc.

12) +double and -double went away; use $$ instead.