The Hybrid Music System
for the BBC Microcomputer

AMPLE Nucleus

PROGRAMMER GUIDE

First published 1987

Hybrid Technology Limited
Unit 3, Robert Davies Court
Nuffield Road
CAMBRIDGE
CB4 1TP

Issue 2

Written by Chris Jordan
Index prepared by Roy Follett

Contents

1 Introduction

AMPLE is the software heart of the Hybrid Music System, and the most powerful music programming environment available on a microcomputer. This Guide is the primary source of information for music programming in AMPLE, covering all facilities of AMPLE Nucleus, the ROM-based core of the system that is common to all applications. Application-specific information, particularly that to do with use of particular hardware units, is deliberately excluded, being reserved for application Guides including the User Guide supplied with your system.

Part 1 - General

2 Using AMPLE

AMPLE is designed to support different hardware installations (combinations of Hybrid Music System units), and different applications of each installation, with corresponding software installations. A software installation may include:

* AMPLE Nucleus – the ROM-based core of the language common to all applications. It provides those functions that are required by all, and manages the modules and user program

* modules – disc-supplied application-specific system units including hardware drivers, user interfaces and extensions for use by the user program

* program – the component which determines precisely what the installation does. In the majority of applications, this is the user's work piece, but in others it is a fixed program suppplied for a particular job.

starting the system

AMPLE is normally started using the system disc supplied for the particular application. The system disc usually has a text file named !BOOT, executed when the user runs the disc by pressing SHIFT+BREAK. This 'boot' file carries out some or all of the following:

1 set system options, including those that may affect the amount of language memory available
2 start AMPLE with the command "AMPLE
3 build the installation by loading modules with INSTALL
4 enter an editor (usually a Main Menu) or load and run a user program

Most system discs do steps 2, 3 and 4.

*AMPLE is the operating system command that enters the language. You are quite free to use it to start the language manually, but since modules will be required for most uses, a boot file is the more usual method.

using the computer keyboard

AMPLE has two general 'modes' that determine how user input is interpreted: **command mode** and **edit mode**.

Command mode is similar in all applications. A '%' prompt appears at the left of the screen when the system is ready for input and an underline cursor shows the current typing position. The user types the line as normal, and pressing RETURN sends it to be executed.

In command mode, most keys on the keyboard have their standard functions, as follows:

printing character keys and space bar
: enter letters, numbers or symbol. The SHIFT, SHIFT LOCK and CAPS LOCK work as standard with these

DELETE
: removes the last character typed

RETURN
: indicates the end of a line of commands

left, right, up, down and COPY
: used as standard for re-entering by copying it from a higher position on the screen

CTRL
: used in combination with other keys to enter special codes, in particular:

  CTRL U — remove line, but leave it on the screen
  CTRL B — turn on screen output to printer
  CTRL C — turn off screen output to printer
  CTRL N — turn on page wait mode
  CTRL O — turn off page wait mode
  When page wait mode is on, the display waits after each screenful of scrolling text unless the SHIFT key is down.
  CTRL SHIFT — hold scrolling text (while pressed)
  CTRL V — does nothing. CTRL V's normal function of changing screen mode is disabled, since this can corrupt language memory on computer models without shadow memory.

f1-f9
: enter a pre-defined sequence of characters. In many applications, the functian keys are defined with commonly-used commands or command segments on start-up. The user can reprogram them with the *KEY command as normal.

The following keys have special functions:

TAB	enter/exit edit mode. Pressing TAB switches between command mode and the edit mode of the editor in use.
ESCAPE	stop everything, and return to the command mode prompt. This is used to stop any AMPLE operation or running program. ESCAPE has other effects specific to the installation – in particular it asks for all sounds to be silenced. See the chapter 'Errors' for details.
BREAK	reset the computer. You should never need to press BREAK in normal use of AMPLE. If you press it by accident, the system will attempt to recover the program. You should save it immediately and then restart the system.

There is a limit on the length of a single input line, normally 128 characters. If you try to exceed this, the computer beeps and ignores the character.

Edit mode is specific to the editor in use. Editors vary widely, but all use TAB to return to command mode. See the application User Guide for details of a particaulr editor's edit mode. See the chapter 'Modules and editors' for general information about editors.

example programs

By convention, if example user programs are provided on the system disc, function key 9 is defined to present them in a menu, so the user can always press f9 after system start-up to see a list of examples. Similarly, any disc with a complete set of user programs, such as a music album disc, uses f9 as the standard method of executing it.

This menu is often a user program called 'jukebox', so function key 9 is defined as follows:

```
*KEY9|"jukebox"LOAD RUN|M
```

Where the system disc starts-up in an editor's edit mode, special provision is usually made so that f9 still operates. For example, the editor may respond to an ASCII 0 input by returning to command mode, and f9 is programmed accordingly:

```
   *KEY9 |@"jukebox" LOAD RUN |M
```

ASCII 0 is ignored in command mode.

screen display

The system uses screen mode 7 for most displays. Most editors work in a specific mode (usually 7) and set this when entered, but command mode can run in any screen mode. You can select the screen mode with:

MODE number	enter display mode	command MODE

On computer models without shadow memory, the screen mode affects the amount of memory available for other uses, so you will normally leave the system in mode 7. There may in fact be insufficient memory free to enter other modes.

Some characters appear with one design in modes 0-6, and another in mode 7:

mode 0-6	name	mode 7 appearance
[	open (left) square bracket	left arrow
]	close (right) square bracket	right arrow
^	'hat'	up arrow
|	vertical bar or 'solidus'	double vertical bar
_	underline	long central line
/	backslash	'half' (1/2)

In this Guide, the normal designs (as modes 0-6) are printed.

operating system commands

In command mode, operating system (OS) commands are entered as standard, prefixed with a '"':

* <line>	indicate operating system command	command

OS commands that corrupt language memory are forbidden. These include those commands that, in their own documentation, are described as destroying the user program, for example *COMPACT & *FORM80 (Acorn DFS), and *FX20 (OS).

Language entry commands act as normal, that is, they leave AMPLE

and enter the corrsponding language, for example:

```
*BASIC
```

The AMPLE Nucleus ROM responds directly only to the command *AMPLE and its abbreviations, such as *AM. It responds indirectly to the *HELP command, giving its name and version number.

AMPLE commands

The following general information applies to all AMPLE commands, including those provided by the Nucleus. Nucleus commands are described individually in the chapter 'Dictionary of words'. See the application User Guide for descriptions of any extra commands.

An AMPLE command is a word that carries out a particular operation when entered at the command mode prompt. Commands are in upper-case letters only – you must enter them in upper case; lower case letters are used for other things.

You can put more than one command on a single line. Each should be separated from its neighbour by one or more spaces to be sure to avoid confusion, though you may find that this is not essential in many cases.

Some special commands actually re-enter command mode, and so any following commands on the same line are ignored. This is made clear in the descriptions of these commands.

Many commands take an input value which may be a string, number or flag. This goes before the name of the command, again separated by a space:

type	format	example
string	characters in double quotes	"myprog" SAVE
number	decimal digits	3 MODE
flag	ON or OFF	OFF PAUSE

A few commands take more than one input value, separated by a space.

Hex and negative decimal numbers are rarely used by commands, but often used by other words:

-ve number	minus sign, then decimal digits	-100 NOUT
hex number	& sign, then hex digits	.. &FFEE CODE ..

See the following in the 'Dictionary of words' for more

information:

```
"  start literal string
&  indicate hexadecimal number
-  indicate negative number
0-9  decimal digits
```

Commands may be abbreviated by a dot. The shortest abbreviation for a given command depends on other commands in the installation, so you should try the intended abbreviation before relying on it.

Some commands respond with error messages if they cannnot act for some reason. These are usually self-explanatory, but for detailed information on a particular error message, see the chapter 'Errors'.

Because AMPLE is a multi-tasking system, you can enter commands at the % prompt while a program is running in the background. Some commands stop the program (usually because they modify the program or its memory) and this is mentioned for each one that does so.

Like commands, programs can produce error messages. These appear on the screen at the command mode cursor (first switching to command mode if in edit mode at the time) and since an error can arise at any time, this could interrupt a line being typed at the time.

Almost all AMPLE Nucleus commands are always available in command mode. In addition, there will be extra commands provided for the particular appplication, described in the application User Guide. Further, the current editor may provide extra commands specific to it – again see the User Guide. The names of all non-Nucleus commands available in each installation and editor can be displayed on command – see the chapter 'Modules and editors' for details.

In special cases, an AMPLE Nucleus command may be replaced in a given installation, or with a given editor in use, by a module providing its own version with the same name. The replacement usually has the same function as the original.

starting a new session

After starting from the system disc, the system is completely clear of user programs and other user data. This is the state from which you should begin a session. To return to it at any time, you use the command AMPLE:

AMPLE	restart system	command

3 Programs and words

programs

An AMPLE **program** is a complete set of instructions for a particular job, for example, playing a piece of music.

In the most general sense, the AMPLE user program is simply the main store for the user's data, and a particular AMPLE user program is the contents of this store. This may or may not be a 'program' in the the traditional sense, that is, a sequence of instructions to carry out a particular job – how the user sees the program depends on the application, and in turn, the user interface employed. For example, the 'program' could be a collection of independent instrument definitions for use with a keyboard. Similarly, a simple multi-part score may look like a 'program' when it is entered by typing, but less like one when created by real-time recording from a music keyboard.

Whatever the application or user interface, the user data is always a standard AMPLE program, on which any of the Nucleus program-manipulation facilities may be used. Further, the application's editors act on the program, and are **integrated** through the ability of one editor to edit a part of the program that was entered with a different one.

program-manipulation commands

The system holds a single program in the computer's memory. The following AMPLE commands work on the program as a whole:

```
   AMPLE  restart system (discarding program)          command
   NEW  discard program, for entry of new program      command

   MERGE  merge program                                command
      namestring MERGE
   SAVE  save program                                  command
      string SAVE
   LOAD  load program                                  command
      namestring LOAD
   WRITE  display text of all words                    command
```

WRITE can also be used to print the program.

words

The basic element of AMPLE is the **word**. Those words that are part of the system are called **system words**, and this type includes all AMPLE commands. Some system words are provided by the Nucleus – these **Nucleus words** are available in all installations of AMPLE. The rest are provided by the installation's modules – they are called **module words**.

The other type of AMPLE word is the **user word** – the basic unit of user programs. User words have names like system words, but they use lower-case letters throughout to avoid confusion with them. Unlike system words, user words can be defined to carry out a variety of functions – the **definition** of a user word is the sequence of other words it performs when executed.

Each word is a separate object which can be created, used, edited and, if necessary, deleted, independently of other words. The user program is the complete set of words considered as a whole.

A user word definition may use any system word that is not 'command only' as a program instruction. Hence, it can carry out a variety of tasks, from active computation tosimple storage, such as for a piece of musical material represented as a sequence of music instructions. Most importantly, any user word can also use any other user word as an instruction, so a program can be constructed using words as building-blocks. For example, a long instruction sequence can be divided into sub-words, to be chained together in a further definition. Further, the program's job can be broken down in to unique tasks which are then defined as words, for example, basic musical material which plays many times in the a complete piece of music.

User words may also be executed as commands, that is, by entering their names directly at the % prompt. In fact, the Nucleus has no RUN command since each program may have many words that can be 'run', either as separate programs or additional commands or options on a single program. In practice, where the program does one particular job, like play a piece, it should include a user word with the name RUN. Some installations may provide a 'starter' RUN word for program development in simple applications, but all finished programs should havs a user RUN word.

word-manipulation commands

The following words define a new user word, or redefine the existing one of the same name:

```
[  start word definition    command
   namestring [...]
]  end word definition      [] only
```

These words are used to define words at three levels:

* direct text – [is used as a command to create a definition by entering it as text directly at the keyboard

* editor text – an editor is used to prepare the text of the definition, and to effectively enter it automatically when it is complete

* editor non-text – an editor is used to enter data in a non-text form which is automatically translated to text for the word definition. The reverse translation is carried out when an existing definition is called up for editing.

The following words are concerned with with user word definitions:

```
SHOW  show user words                                  command
DELETE  delete user word                               command
   namestring DELETE
TYPE  type the word definition on the screen           command
   namestring TYPE
RENAME  rename word                                    command
   oldnamestring newnamestring RENAME
FIND find uses of word                                 command
   namestring FIND
```

user word formatting

Carr iage returns (line separators) and spaces between instructions may be used inside definitions to make the text easier to read, and comments – textual notes of your own which are ignored by the system - may also be included.

```
% introduce comment
<carriage return> mark line end
<space> separate items
```

memory usage

The following words are related to the use of memory space for the program and other items:

MEM show memory usage in bytes	command
CDMPACT compact unused memory	command
MODE enter display mode	command
number MODE	

4 Modules and editors

Modules are the RAM-based extensions to the Nucleus that provide those facilities that are specific to a particular installation. Each module interfaces the Nucleus with one of the following:

* hardware – drivers for peripheral units, including for musical voices ('voice servers'), for musical time ('time servers')
* user – editors, menus, and other user interfaces and command utlilites
* program – predefined instruments, music actions, programming extensions, etc.

Each application's User Guide describes the modules included on its system disc. This chapter gives a general description of the module system, and introduces the Nucleus words that relate to it.

module functions

Modules provide the following types of function in the system:

* voice server – controls a music peripheral to provide music voices which have a voice assignment word and support the standard set of voice controls (described in the chapter 'Music and sound').

* time server – controls the passage of musical time by supplying a timebase signal to the Nucleus, derived from internal or external hardware (described in the chapter 'Music and sound').

* editor – provides an alternative user interface, usually to allow word definitions to be created and altered in a particular form, such as text, staff notation etc., but also to provide a menu of command options (described below).

* command utility – supplies additional commands for special applications such as advanced program and word manipulation. The extra words are available at all times when the module is present and are used exactly like Nucleus command words.

* extension – extends the vocabulary of words for use in programs, for special applications. The new words are available at all times when the module is present, and they

are used exactly like Nucleus words. Examples include 'preset' instruments (for a particular voice type), graphics drivers, and some of the functions given above, particularly voice servers.

A module does not have to be only one of these, but usually each functional unit is supplied as a single module.

examining modules

The following words allow modules to be examined:

```
MCAT  display catalogue of modules                command
MSHOW  show words in module                       command
   modnamestring MSHOW
```

MCAT displays the status of each module. The status affects the operation of the other module management commands.

module names

Each module has a short upper-case name, displayed by MCAT, and it is stored on disc as a file with the same name. By convention, modules are stored under directory 'M' (for example, the full filename of the module INT would be M.INT), and the !BOOT file uses the following command to inform the Nucleus of this:

```
MPREFIX  set module filename prefix                command
   string MPREFIX
```

Every module load operation adds the MPREFIX string to the start of the module name to make the filename.

The main filename of a module must not be changed, but the prefix can be, provided the system is informed through MPREFIX.

A system disc often includes a drive specifier in its MPREFIX string. This directs module loads to a specific drive (usually the auto-boot drive, drive 0) so that the default drive selection can be set for user program filing.

loading modules

Modules are loaded by the following commands:

```
   INSTALL  install module                              command
      namestring INSTALL
   MLOAD  load module                                   command
      namestring MLOAD
   LOAD  load program                                   command
      namestring LOAD
```

module loading on start-up

When the language is entered with the command *AMPLE, no modules are present. In this state, AMPLE is entirely usable, but the only words available are those of the Nucleus.

A typical system disc starts the system by entering the language with *AMPLE, and then loading each module with the INSTALL command. Each module becomes a fixed part of the installation, and cannot be removed without restarting with *AMPLE.

The set of INSTALLed modules includes the following kinds of module:

* essential hardware drivers, such as voice and time servers, which will probably be needed in every session.

* essential editors and menus, such as a main menu which loads editors as required

Some modules are inherently 'fixed only', so that they always become fixed on loading, however they are loaded.

The system disc can be configured to load any module on start-up, simply by adding an appropriate INSTALL command to the !BOOT file. The user could, for example, INSTALL a commonly-used editor or extension module to save the extra memory used by MLOADed modules. To avoid disturbing an existing INSTALLed module thay may require a specific memory range, additional modules should be loaded after the last existing module. Equally, the user could remove a module from the INSTALL sequence to save memory, either because it was not required at all, or was better loaded when required.

module loading by the user

Editor and command utility modules are often loaded by the user when required, and then deleted when finished with (similar to code 'overlays' in other systems). Loading is either direct, by the user entering an MLOAD command, or indirect, through a menu which uses MLOAD internally.

In most installations, a menu looks after such temporary module loading. This 'main menu' is itself provided by a module which is installed, so available at all times. The menu provides a list of available editors and utilities, and records which one it last loaded. When the user makes a selection, the menu carries out the following:

* if the selected editor/utility is already present, it invokes it, else it:
* deletes the module last loaded
* loads the selected module
* enters the editor/utility

Note that the module is invoked without loading if already present for any reason, so the user can get faster access to a module by INSTALLing it in the !BOOT file or first loading it manually with MLOAD.

module loading by the program

Each saved program contains a list of the modules which it needs – a program needs a particular module if it uses any word from it as an instruction in a word definition. When a program is loaded, the LOAD command automatically MLOADs any needed modules that are not already present. When the program is deleted by NEW or a further load, the program loaded-modules are automatically deleted. This powerful feature allows a program to use extension modules without the user having to be aware of it.

module deletion

The following words delete modules:

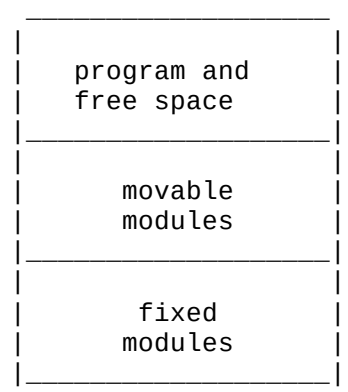
```
MDELETE  delete module                          command
   namestring MDELETE

LOAD  load program                              command
NEW discard program                             command
AMPLE  restart system                           command
```

reading module word definitions

Some module word definitions may be read by the user, for example, displayed with TYPE or called in to an editor. Preset instrument modules are usually of this type, allowing the user to create a modified version of an instrument as a user word definition.

module memory usage

Modules are held in a single block of memory along with the user program (word definitions and public data) and free space, as follows:



Both internal boundaries are movable. Modules are loaded from the bottom up, and when a module is loaded, the amount of memory available for the program is reduced accordingly.

Each module file consists of the module proper, plus relocation data which is needed to position the module in memory. When a module is loaded by INSTALL at start-up (or a 'fixed-only' module is loaded by any means), it is put in place and the now-redundant relocation data discarded to save memory space. Its MCAT status is 'F', for fixed.

When a module is later loaded by MLOAD or LOAD, its relocation data is retained, so it can be moved down when a previously-loaded module is removed, reclaiming space for use by the program. An exception is made for the first module, that is, the one immediately above the last INSTALLed module – Since it will never need to be moved, its relocation data is always discarded.

Points to note from this are:

* any module can be INSTALLed rather than MLOADed in order to

save memory space
* the amount of user program memory depends on the order in which two movable modules are loaded
* all modules below a fixed module are also fixed – INSTALL is used only at start-up, and the 'fixed-only' property is merely a precaution to prevent removal.

editors

The common features of all AMPLE editors are:

* an 'entry' word, usually the name of the module, which makes this editor the current editor
* an 'auxillary dictionary' – a list of commands that are only available when the editor is current (displayed by MSHOW)
* a 'tab' command (on the auxillary dictionary), which enters the editor's 'edit mode'.

There is only one current editor at any time, so entering an editor automatically exits the previous one. MCAT shows which module is the current editor. Many editors create special screen displays when entered, and clear them when exited. There is also a Nucleus command which exits the current editor by switching to a 'dummy' editor:

```
QUIT  leave editor                                    command
```

The command AMPLE executes QUIT.

Entering an editor may leave control back in command mode, or in the editor's edit mode.

editor types

A typical editor offers the following facilities:

* a data area in which the data currently being edited is held
* an edit mode in which the data is displayed and may be edited. Pressing TAB returns to command mode
* a command mode with the folowing additional commands:
  * the entry word, to enter the editor, leaving control in command mode (the % prompt) or edit mode
  * CLEAR to clear the data area
  * GET to get the contents of a named word in to the data area
  * MAKE to define a word with the contents of the data area
  * NAME to set the name of the word to be created by MAKE
  * 'tab' to enter edit mode

(The tab command is a command like any other, but beause the % prompt ends the line on a TAB press, the command is executed immediately. The tab command does not show on the MSHOW display.)

Generally, editors may have more or less command-mode facilities that these. In particular, simpler user interfaces like menus are sometimes provided as editors, but they may have no more than an entry word and 'tab' command – a very simple edit mode displays the list of options and waits for a selection to be made, and then returns to command mode as soon as the selection is executed.

editor data

Normal editors manage their own data area, so their data is private and cannot be accessed except by that editor. Advanced editors use 'public' data which can be accessed directly by other editors, and is held as part of the user program so it is saved and loaded along with the word definitions.

Public data has a 'type' associated with it, displayed by the SHOW command. The commonest type is text, displayed as 'T'.

Public data can be edited with any editor that recognises the type. When a public data editor is entered, the data is immediately available for editing if the type is compatible, and it is cleared if it is not compatible.

The Nucleus has a command that clears the data explicitly:

```
   CLEAR  clear editor data                              command
```

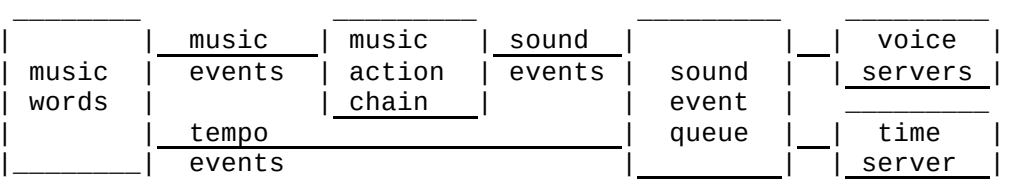
After CLEAR, SHOW displays 'no data'. The type is set when data is next provided by an editor or loaded with LOAD.

A private data editor will often provide its own CLEAR command which is used in preference to the Nucleus CLEAR.

5 Music and sound

introduction

AMPLE's complete music and sound system can be illustrated as follows:



The components of this system are now discussed individually.

music words

AMPLE Music Notation is the textual music representation provided for scoring of note-based music. It is used both directly by the user, and as a standard internal form by editors providing other notation forms, such as staff notation.

AMPLE Nucleus supports AMPLE music notation by incorporating all necessary music words as a self contained set, and providing a standard interface between this and the voice and time control systems.

The music words are of two types: music event words and music environment words. Music event words cause direct musical results, according to values set by the music environment words.

music environment words

The following words control the music environment:

```
Length:   ,  set length
             number ,
          BAR  set bar length
             lengthsnumber BAR
          |  mark end of bar

Pitch:    :  set octave
             octnumber :
```

```
          !  move an extra octave up or down

          +  sharpen next note
          -  flatten next note
          =  naturalise next note
          K( start key signature
          )K end key signature

          @  set transposition
             transnumber @

Voice:    (  start additional chord notes
          )  end additional chord notes
          ;  set music voice voicenumber

Gate:     ~  slur next note

Level:    =L set dynamic level
          +L increase dynamic level
          -L decrease dynamic level

The following also have an effect on the music environment:

   SCORE  prepare for music words
   READY  ready system

The following access the music environment values for processing:

   MVAL? read music variables
      MVAL? -> framelev keysig barcountlen octnote length tranvoice
   MVAL! write music variables
      framelev keysig barcountlen octnote length tranvoice MVAL!
```

music event words

```
Voice events:    A – G  play ascending note
                 a – g  play descending note

                 X  play hit

                 ^  play rest
                 ^; play chord rest

                 /  hold notes, hits, and rests
                 \  move back

                 (  start additional chord notes
                 )  end additional chord notes
```

(The chord words '(' and ')' are considered music environment words from a notational point of view, but in fact they generate simple music events, so are strictly-speaking music event words.)

```
Tempo events:    =T  set tempo
                    number =T
                 +T  increase tempo
                 -T  decrease tempo
```

music interpretation

The interpretation of music notation is the conversion of high-level music events such as 'note', 'rest' etc. into low-level sound events having precise effects like 'set pitch', and 'wait for a period of time'.

AMPLE interprets the two types of music event words – voice and tempo – separately.

Each tempo event word simply generates its own type of tempo event, which includes a single variable value. This event passes via the sound event queue to the time server where it takes effect on the timebase, the regular repeating pulse that marks the overall passage of musical time. All durations, including the lengths of notes etc., work in timebase units, so the timebase controls the final duration of all notes and therefore the tempo of the music.

All voice event words (note letters, etc.) generate a single type of music event with some or all of the following variables:

voice	voice selection
pitch	pitch in semitones
'vel'	dynamic level, or 'key velocity'
gate	state: on (sounding) or off (silence)
duration	period of time, in timebase units

Each music event is completely defined by the values of these five variables. They are carried by sound events, via the sound event queue, to a voice, where they produce the sound of the original music event.

The voice variable is given to the sound word VOICE, to select the voice to which subsequent sound events will be directed. The pitch, 'vel' and gate variables are given to PITCH, VEL and GATE respectively – standard sound words which all voice types have. The duration variable is given to the sound word DURATION, which adds a final time interval through its effect on the sound event

queue itself.

(In fact, the music event has three separate voice values for PITCH, VEL and GATE; not so that they can be sent to different voices, but be individually excluded by a voice value of 0, if it is not required in that music event.)

Notice that all music events, and notes in particular, are 'single ended' – an initial group of sound events marks the start of a note, a duration gives it length, but nothing marks the end of it except the start of the next music event.

music actions

AMPLE gives powerful control over music event interpretation through the use of **music actions**. A music action is a sequence of instructions that is executed every time a music event is issued. It can use any Nucleus programming words and access the music event variables to modify, transform, augment or replace the standard interpretation. Music action definitions are often provided by extension modules, and may be created by the user through the following words:

```
ACT  execute music action
ACT(  start music action sequence                  [] only
   positionnumber ACT(
)ACT  end music action sequence                    [] only
SIMPLEACT  remove all music actions
```

The folowing number processing words are used to access the music event variables in music action definitions:

```
FVAR  access stack frame item
  itemnumber FVAR -> addressnumber
VOICE! change voice settings in frame
   voicenumber VOICE or ON VOICE!
```

sound events

Information is carried from the user program to the low-level music peripheral driver software in the form of **sound events**. A sound event is a simple instruction to set the value of a particular control on the receiver. There are four main types:

* voice events
* type-global voice events
* time control events (including tempo events)
* queue control events

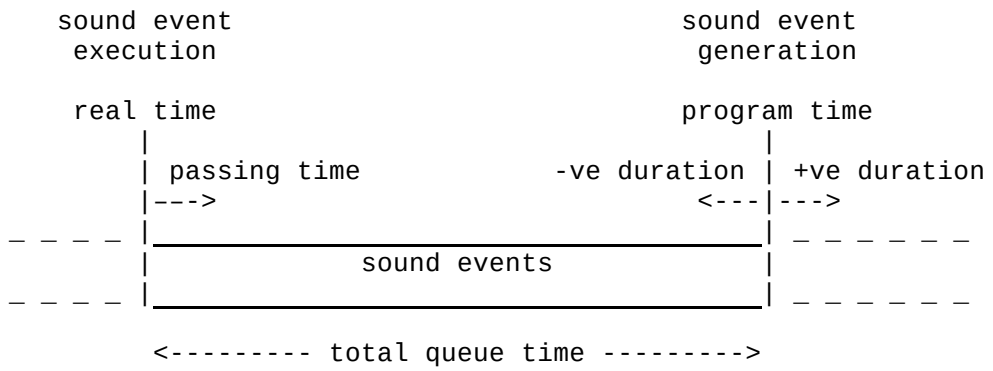
Sound events are generated by sound words. The Nucleus provides queue control sound words, and voice servers provide their own voice sound words and type-global voice sound words. Tempo sound events are generated only by tempo music words, so no tempo sound words are provided.

the sound queue

When a sound event is generated by the user program, the system does not execute it immediately, but stores it temporarily on the **sound queue**, a buffer in which the events are held in the order they are to be executed, separated by durations. Durations are continuously consumed from the 'output' end of the queue under the control of the timebase, and as sound events are encountered, they are vemoved and executed. All operations carried out by sound events are therefore synchronised to the timebase. This advanced mechanism provides two important benefits:

* sound execution is precisely timed, and independent of execution of the generating program
* sound events may be generated in an order independent of the order in which they are to be executed.

The sound queue represents a segment of the total duration of a piece of music, storing all the sound events required to play in that period. This segment continually moves forward as passing time consumes it at the 'real time' (time present) end and the program extends it at the 'program time' (time future) end:



Durations move program time, and 'total queue time' is a measure of program time, relative to real time. The following words provide these functions:

```
   DURATION  wait for a period of time
      number DURATION
   QTIME  return queue time
      QTIME -> number
```

Two further words have direct effects on the sound queue:

```
   READY get ready for players
   GO  start players together
```

Each player has its own independent program time.

The system command line interpreter resets player 0's queue time before executing each line. See QTIME for details.

queue control sound words

The following words generate sound events which control queue processing itself:

```
   FAST  select fast/normal tempo
      flag FAST
   PAUSE  pause/continue sound processing
      flag PAUSE
```

time control

AMPLE Nucleus uses an external **time server** (supplied as a module) to provide the timebase and thereby control the passage of musical time. This allows the method of time control to be changed to suit the application. Most installations will include a time server providing a free-running, computer-internal timebase, whereas others might accommodate external sources, such as for synchronisation with external equipment.

Only one time server is active at a time, but more than one time server module can be installed. To select between alternative time servers, each one has a word, usually the same as its name and having no input or output values, which selects it and deselects the previously selected time server.

The current time server responds to tempo control events generated by the tempo music words, and other time control events generated by the following:

```
   WIND  advance time
      ticksnumber WIND
```

```
   HALT  halt/continue timebase
```

voice selection

AMPLE Nucleus can accommodate 132 voices, 12 in each of 11 'ensembles'. In practice, the installation provides a smaller number of voices, but these can be individually assigned to any of the 132 voice positions, allowing great flexibility in voice use.

A voice position is identified by the number of the ensemble and the number of the voice within the ensemble. In most cases, the ensemble number corresponds to the player number, because the system selects the same-numbered ensemble for each player on its creation.

Each player has a voice selection which determines which voice or voices its subsequent voice sound events affect. This selection consists of the number of the ensemble and the number of the voice within the ensemble, and these are set using:

```
   VOICE  select voice(s)
      voicenumber VOICE
   SHARE  select voice ensemble
      ensemblenumber SHARE
```

When a player is created, it automatically has the same-numbered ensemble selected, so ensemble 1 'belongs' to player 1, ensemble 2 to player 2, etc. SHARE can then be thought of as selecting (or 'sharing') the specified player's voices. Often, each player addresses only its own voices, and SHARE is not used.

Two further words set the 'range of voices' on the addressed ensemble (not player). The complete range can be selected by VOICE so that each sound event is automatically sent to two or more voices.

```
   VOICES  set number of voices
      number VOICES
   RVOICES  set voices range
      startnumber endnumber RVOICES
```

voice assignment

Each voice type has a **voice assignment** word which assigns a voice of that type to the selected voice position or positions. Only once a voice is assigned to a position can it receive further sound events and hence be used.

The Nucleus has a special 'unused' voice type, assigned by:

```
UNUSED  make voice(s) unused
```

voice events

Each voice type has a set of sound words that give access to the voice controls, through sound voice events. They include basic 'performance' controls like pitch and gate, and 'instrument' controls that are designed for selecting or defining overall characteristics. In principle, any control could be used for performance or instrument definition and many voice types do allow this, but some place restrictions on the use of certain controls, particularly if their voices are on remote, rather than integrated, devices.

In addition to a voice assignment word, all voice types have the standard sound words required for default interpretation of AMPLE Nucleus music words:

```
PITCH  set pitch in semitone units from middle C
   pitchnumber PITCH

VEL  set dynamic level in range 0 to 127
   levelnumber VEL

GATE  set gate state, 'on' or 'off'
   gateflag GATE
```

The precise interpretation of these varies between voice types and event instrument definitions on a single voice type, but the overall effect is as defined by the words' descriptions.

type-global voice events

These affect all voices of the same type together. The corresponding sound words are specific to the voice type and are not affected by the voice selection. Examples include global tuning and volume.

voice servers

A voice server is a module that provides a voice type, and drives a particular type of voice-providing music device. It provides the sound words for its type and the low-level routines that implement logical voices in terms of the physical voices available from the device.

6 Numbers and flags

AMPLE supports **16-bit signed integers**, that is, whole numbers in the range -32768 to 32767.

A literal number is one that appears where its value is required, and in AMPLE may be in decimal, negative decimal or hexadecimal form.

```
& indicate hexadecimal number
- indicate negative number
```

arithmetic expressions

AMPLE uses **post-fix** notation for all operations on numbers. This is a contrast to languages like BASIC that use **in-fix** notation. In BASIC, arithmetic expressions appear in mathematical style, with each two-input operator appearing between its input values, for example

```
2 + 3       or      4 * (2 + 3)
```

Sub-expressions are bracketed to define the order of the operators.

In AMPLE, expressions work in computer style, with numbers and operators appearing in the order in which they act, that is, with the operator after its input values, for example:

```
2 3 #+      and      2 3 #+ 4 #*
```

(AHPLE uses '+' for a musical function, so the arithmetic add instruction is ' #+').

Because AMPLE executes the items in strict left-to-right order, brackets are never required for sub-expressions. In fact, because numbers and operators are executed step-by-step like any other program instructions, AMPLE expressions are much more flexible and often simpler than their BASIC equivalents, even though they may look less familiar.

The following arithmetic operators are available:

```
#* multiply two numbers
  number1 number2 #* -> productnumber (number1 x number2)
```

```
   #+  add two numbers
      number1 number2 1+ -> sumnumber (number1 + number2)
   #-  subtract number from previous number
      number1 number2 #- -> differencenumber (number1 – number2)
   #/  divide previous number by number
      number1 number2 #/ -> quotientnumber remaindernumber
   #B12  swap high and low bytes of number
      number1 #B12 -> number2

   MAX  leave larger of two numbers
      number1 number2 -> largernumber
   MIN  leave smaller of two numbers
      number1 number2 -> smallernumber
```

A useful word for experimenting with expressions is:

```
   NOUT print number in decimal
      number NOUT
```

for example:

```
   2 3 #+ NOUT   prints      5
```

constants

A numeric constant is an item that has a fixed numeric value. In AMPLE, this is simply created as a word containing a literal, for example:

```
   "semiperoct" [ 12 ]
```

The name of the word can then be used anywhere that a literal is allowed, for example, in an expression:

```
   3 semiperoct #+     is equivalent to        3 12 #+
```

Using a constant instead of a literal makes it simple to change the value once incorporated into a program, especially if the value is used in two or more places. It can also make the program easy to read by describing its function, and in addition a comment may be included:

```
   "semiperoct" [ 12 % number of semitones per octave
   ]
```

the number stack

AMPLE uses a structure called a **stack** to hold numbers as it works through an instruction sequence. This is effectively a pile to which items may be added and removed at the top position only – a last-in first-out buffer, or 'LIFO'. You don't have to know about the stack to use numbers and expressions, but it helps in understanding more advanced uses.

A literal number simply puts its value on the stack, for later use by an operator. Each operator takes its input values from the top of the stack, and leaves its output value on the top of the stack. Some words, like NOUT, simply consume a number, that is, they have no output.

A literal number may remain on the stack to be consumed by an operator in another word, as illustrated by our definition of a constant. A constant is an example of a word passing out a value: one that has no input values and one output value.

The stack can be used for temporary storage, since each number is unaffected by the activity above it, for example:

```
1                  % leave on stack
2 3 #+ NOUT        % prints     5
NOUT               % prints     1
```

(This sequence can only be executed from within a word definition, since the system will give the 'Extra number' error on finding the unused '1' at the end on the first line, and then the 'No number' error on reaching NOUT.)

A simple example of temporary storage is:

```
1 2 NOUT SP NOUT    % prints   2 1
```

Of course, the numbers are reversed – last-in, first-out.

Note that each player has its own number stack, so number processing and storage is entirely independent for each player. Each number stack can hold 31 numbers.

passing numbers

Since the stack lets numbers be freely passed in and out of word definitions, an expression can be put into a word definition, and then receive a value from outside, for example:

```
"printsemi" [ 12 #* NOUT ]
2 printsemi

is equivalent to

2 12 #* NOUT

and prints    24
```

This structure is the equivalent of a **procedure** in other languages: a named sequence of instructions supplied with input values each time it is called.

It is conventional to indicate the number of input values required, with a comment inside the word, particularly since this information is not necessarily obvious from the form of the definition. The comment shows the form of use of the word, in the style used in this Guide, for example:

```
"printsemi" [ % octnumber printsemi
... ]
```

A **function** is simply a procedure with one output value, such as:

```
"tosemi" [ % octnumber tosemi -> seminumber
12 #* ]
2 tosemi NOUT

is equivalent to

2 12 #* NOUT
```

An AMPLE function may have more than one output value. The divide operator, #/ , is an example:

```
#/  divide previous number by number
   number1 number2 #/ -> quotientnumber remaindernumber
```

The input/output description tells us that

```
   7 3 #/      is equivalent to        2 1
```

so

```
   7 3 #/ NOUT SP NOUT        prints   2 1
```

as does

```
   1 2 NOUT SP NOUT
```

It should be clear to you that, in AMPLE, literals, constants, operators, procedures and functions are all just variations on a simple theme, and they can be created as simple word definitions without any 'red tape'. You may also have noticed that there is no need for any sort of statement separator, since execution always progresses logically through the instructions as they appear. However, you can make the program clearer to read by laying out groups of instructions on separate lines, or with extra spaces in-between.

stack operators

The stack operators allow you to take special advantage of the stack's storage abilities by duplicating, discarding and rearranging the top numbers.

```
   #11 duplicate number
      number #11 -> number number
   #12  swap two numbers
      number2 number1 #12 -> number1 number2
   #2 discard number
      number #2
   #212 duplicate previous number
      number2 number1 #212 -> number2 number1 number2
   #2121 duplicate number and previous number
      number2 number1 #2121 -> number2 number1 number2 number1
   #213 rotate positions of three numbers
      number3 number2 number1 #213 -> number2 number1 number3
```

The names of these words directly describe their net effects on the following hypothetical stack:

```
9 8 7 6 5 4 3 2 1
```

For example:

```
9 8 7 6 5 4 3 2 1 #11       produces
9 8 7 6 5 4 3 2 1 1
```

and

```
9 8 7 6 5 4 3 2 1 #213     produces
9 8 7 6 5 4 2 1 3
```

A further set of words allow access to numbers at specifiable positions, or **frames**, on the stack, independent of the number of items above:

```
FCOPY  copy numbers from frame
   number FCOPY
FRAME mark top of stack frame
FRAME!  set frame pointer
   pointernumber FRAME!
FRAME?  read frame pointer
  FRAME? -> pointernumber
FVAR  access stack frame item
   itemnumber FVAR -> addressnumber
VOICE! change voice settings in frame
   voicenumber VOICE or ON VOICE!
```

flags

A flag is a logical value: one that is either ON or OFF. AMPLE represents flags as numbers, with 0 standing for OFF and -1 for ON. Flags and numbers are equivalent as far as the system is concerned, so all words described as number operators can equally be used on flags.

The following words produce flag results:

```
OFF  leave false flag
   OFF -> offflag
ON  leave true flag
   ON -> onflag

#<  test previous number is less than number
   number1 number2 #< -> flag (number1 < number2)
```

```
   #=  test numbers are equal
      number1 number2 #= -> flag (number1 = number2)
   #>  test previous number is greater than number
      number1 number2 #> -> flag (number1 > number2)

   SIGN  test number is negative number
      SIGN -> flag
```

flag operators

The following flag operators work on individual binary bits of the numbers, and they can be used either on flags as logical connectives, or on numbers as 'bit-wise' operators.

```
   AND   AND bits of numbers
      number1 number2 AND -> ANDnumber (number1 AND number2)
   OR  OR bits of numbers
      number1 number2 OR -> ORnumber (number1 OR number2)
   XOR  exclusive-OR bits of numbers
      number1 number2 XOR -> number3
```

NOT is a strict logical connective, meaning that it does not operate bit-wise.

```
   NOT  invert sense of flag
      flag1 NOT -> flag2
```

variables and storage

As you may expect, variables and arrays are also defined as simple words.

```
   GVAR create variable                                [] only
      GVAR -> addressnumber
   DIM  reserve memory
   ARRAY  access array element
      elementnumber baseaddressnumber ARRAY -> addressnumber
```

A variable or memory block is defined like any other word, using the appropriate instruction inside the definition, for example:

```
   Variable:       "total" [ GVAR ]
   Array:          "totals" [ 100 DIM ARRAY ]
```

The storage space for the variable is provided by the GVAR instruction itself, but the DIM space is claimed from language memory, and is accounted for in the 'Arrays' entry of SHOW. DIM

may be used without the ARRAY add-on to reserve a simple block of memory for direct access by the program, for example

```
"buffer" [ 100 DIM ]
```

Other items, such as comments, additional actions, debugging aids etc. can also be included in any of these definitions.

GVAR and DIM definitions return absolute memory addresses for the use of the following store and fetch operators:

```
#!  store number at address
   datanumber addressnumber #!
#?  fetch number from address
   addressnumber #? -> datanumber

#+!  add number to number at address
   datanumber addressnumber #+!

#B!  store low byte of number at address
   datanumber addressnumber #B!
#B?  fetch byte from address
   addressnumber #B? -> datanumber
```

These are suitable for flags as well as numbers.

From the point of view of a BASIC programmer, #! is the post-fix equivalent of the assignment operator '=':

```
BASIC:      total=1
AMPLE:      1 total #!
```

In an expression (for example, at a position to the right of a BASIC '='), each variable is followed by #? to 'fetch' its value, for example:

```
BASIC:      total=input
AMPLE:      input #? total #!
```

Because AMPLE is not restricted to formal 'statements', operations can often be carried out with fewer instructions than in BASIC, for example:

```
BASIC:      t = x : x = y : y = t           :REM swap x and y
AMPLE:      x #? y #? x #! y #!             % swap x and y
```

random numbers

The following words control AMPLE's built-in random number generator:

```
RAND  get random number
   RAND -> number
RAND!  set starting point for random numbers
   number RAND!
RANDL  get random number in range
   maxnumber RANDL -> number
```

7 Characters and strings

Single characters may be are represented as numbers, for example, a letter by its ASCII code. Hence, the number stack and number operators may be used for processing single characters.

Strings are supported as a separate type, using a single string stack similar in operation to the number stack. A string is a sequence of up to 128 characters, including all 8-bit values. The following words convert between a character on the number stack and a one-character string on the string stack:

```
ASC  convert string to number                        [] only
   string ASC -> asciinumber
$CHR  convert number to string                       [] only
   asciinumber $CHR -> string
```

The number value -1 represents a null string/character.

Literal strings may be included in the program by enclosing them with " characters, for example:

```
"hello"

"  start literal string
   "characters" -> string   (when inside [ ... ] : see later)

A " character may be included in the string by repeating it.
```

string operators

The following string operators are provided:

```
$+  add string to left end of previous string        [] only
   rightstr ing leftstring $+ -> string (left + right)
$-  split string after numbered character    [] only
   string number $- -> rightstring leftstring
$PAD  pad string with spaces        [] only
   string1 lengthnumber $PAD -> string2
$REV  reverse the order of characters        [] only
   string $REV -> reversedstring
$STRIP  remove leading spaces       [] only
   string1 $STRIP -> string2

LEN get length of string    [] only
```

```
     string LEN -> string lengthnumber
```

The string stack operators are:

```
   $12 swap two strings        [] only
      string1 string2 $12 -> string2 string1
   $2  discard string  [] only
      string $2
```

Conversion between number and string form is provided by the following:

```
   $STR  convert number to decimal string represenation [] only
      number $STR -> string
   &$STR  convert number to hex string representation  [] only
      number &$STR -> string
   VAL  convert string to unsigned decimal number      [] only
      string VAL -> remainingstring number ON    if found
      string VAL -> remainingstring OFF          if not found
   &VAL  convert to unsigned hex number        [] only
      string VAL -> remainingstring number ON    if found
      string VAL -> remainingstring OFF          if not found
```

string stack usage

As an example of normal usage, a word can simply leave a string with "..." and then print it out immediately:

```
   "speak" [ "hello" $0UT ]
   speak                        % prints    hello
```

Equally, it could leave a string to be picked up by another word:

```
   "it" [ "hello" ]   "say" [ $OUT NL ]
   "speak" [ it say ]
   speak                        % prints hello
```

Whereas you may pass a string from instruction to instruction in a definition as this example shows, you may not pass a string from command to command, that is, along the command line. This is because the command input line is itself held, by the system, as a string on the string stack. For example, if in the above example, 'it say' was entered as a command sequence, the data string would interfere with the input line, producing an error:

```
   speak                        % prints      hello
   it say                       % fails with '! Mistake'
```

Remember, there is no restriction in normal string stack usage

inside word definitions, that is, usage in which any word executed from the command line preserves the state of the string stack

using strings at the command line

Because the command input line is itself a string on the stack, " has a different action when outside [...], and many string operators are restricted to use inside [...].

Whereas "..." puts the string on the top of the stack when inside [...], when entered in the command line, it puts the string as the second item down, that is, underneath the input line string. So, the complete stack action of a literal string (the word ") is actually:

```
   "characters" -> string                              inside  [...]
   inputstring "characters" -> string2 inputstring     outside [...]
```

System commands that take strings (SAVE for example) take them from below the input line string with $12, leaving the input line string undisturbed as the top item.

User words can accept direct-mode strings in the same way, for example:

```
   "say" [ $12     % get direct-mode string from under input line
           $OUT ]
   "hello" say     % prints    hello
```

User words can deliver direct-mode strings also by using $12, for example:

```
   "name" [ "program" $12 ]
   name SAVE       % equivalent to   "program"SAVE
```

using the input line

As the system's command interpreter works along the input line string, it finds the next word name, removes it, and executes the word. The last word on the line is either carriage return or TAB (carriage return does absolutely nothing unless inside [...]). When the input string is entirely consumed, that is, reduced to "", the system prints the % prompt and waits for the next command line to be entered. At the point of executing any word on the line, the input line string is the remainder of the original line, that is, the part of the line to the right of the word being executed. Certain system words like & and % access the input line directly to perform their functions, and user words can do the

same for special purposes.

By adding to the input line string, a user word can issue commands, for example:

```
"memshow" [ "MEM SHOW" $+ ]
memshow                    % equivalent to   MEM SHOW
```

When 'memshow' starts executing, the top (and only) string on the stack is the remainder of the input line; in this case, a single carriage return. 'memshow' adds MEM SHOW to the start of the string, and then finishes. Control returns to the command interpreter which finds the string MEM SHOW<CR> as the remaining input line, and processes it exactly as if it had been entered directly by the user.

Commands such as this that substitute the name or names of other commands to achieve their functions are called 'macro' commands or just 'macros'. This example is a simple macro that produces the same output string every time. Computed macros are more advanced, producing an output dependant on input values, variables, user input etc. An example of a computed macro is a user menu made with menu construction words supplied in most installations – when the name of the menu word is entered as a command, it replaces itself by a command sequence that is selected by the user.

A command word can access items following it on the input line by simply examining the input line string, and modifying it if required. A very simple example is:

```
"say" [ $0UT "" ]   % print remaining line, and replace by ""
say fred            % prints fred
```

'say' consumes the whole of the remaining input line, including the final carriage return, and replaces it by "" (it does not need to replace the carriage return).

The following example decodes a number from the input line – it is a BASIC-style MODE command which is used with the number after it, e.g. mode 7.

```
"mode" [
$STRIP      % remove leading spaces
VAL         % get number from input line
#2          % discard VAL flag (ON if number found)
MODE ]
mode 7 MEM  % change to mode 7, and do MEM
```

If no following number is found, 'mode' takes a number from the stack anyway, so in fact the command can be used either way

around: mode 7 or 7 mode .

It is important to remember that a user-defined command word always acts as a command word, working on the current input line. If you use a command word inside [...], then the defined word is also a command word. For example, you can define a new command as follows:

```
"modecat" [ mode "CAT"OSCLI ]
modecat 7           % give mode 7 catalogue display
```

What you cannot do is include the complete command as an instruction inside a word definition, for example:

```
"teletext" [ mode 7 ]
```

Entering 'teletext' will not have the same effect as entering 'mode 7'.

using strings in players

The single string stack is common to the whole system, that is, it is shared by all players, so when two or more players want to use the string stack independently, a simple restriction applies. ('Two or more' includes the static player).

Basically, only one player can leave strings on the string stack permanently – all other players must preserve the state of the string stack over player control transfer points. These points, called 'idle' points, are present at any instruction from which control may not immediately return to the player. They include:

```
IDLE
#IN, $IN
A-G, a-g, X, /, ^, (, ) (all music events)
ACT, DURATION, HALT, FAST, ON PAUSE
+T, -T, =T
P(
sound words (PITCH, GATE etc)
```

Where a dynamic player runs alongside the static player's command interpreter, it should always preserve the string stack state, since the command interpreter uses the string stack to hold the input line.

string stack capacity

The string stack has a total capacities of 128 characters and 16 strings.

The character capacity available in a word definition is reduced by the number of characters remaining on the command input line, so an otherwise successful definition could fail if executed from the start of a longer command line.

Programs that make heavy use of the string stack may guard against this by discarding the input line on entry, and replacing it by a null string on exit:

```
"RUN" [ $2  % discard input line
...         % run program
"" ]        % leave null input line
```

This also makes all of the stack's 16 string positions available to the program.

8 Input and output

AMPLE Nucleus provides a variety of words for QWERTY keyboard input and screen output via the standard operating system interfaces. In addition, the user may access operating system input and output routines directly, for special applications.

numbers

Words for number output are also provided:

```
NOUT  print number in decimal
   number NOUT
&NOUT  print number in hexadecimal
   number &NOUT
```

characters and strings

The following words handle character input and output:

```
#IN  wait for and get keypress
   #IN -> asciinumber
QKEY  test key status or get keypress
   negativenumber QKEY -> flag
    zeronumber QKEY -> asciinumber
#OUT  send ASCII code to screen
   number #OUT
```

String input and output are provided by:

```
$IN  input line from keyboard                        [] only
   $IN -> string
$OUT  print string                                   [] only
   string $OUT
```

Number input can be implemented with string input and VAL/&VAL for string-to-number conversion. See VAL and &VAL for examples.

The following provide special output to the screen:

```
ALIGN  ensure text cursor is at start of line
MODE  enter display mode    command
   number MODE
NL  print new line
```

```
   SP print a space
```

system effects

AMPLE automatically sets various keyboard and screen options when terminal-style line input is called for. The following settings are made by $IN and the command line before accepting input:

```
   *FX255, 1                      % make function keys expand
   *FX4,0                         % engage cursor editing mode
   OSWRCH: 23 1 0 0 0 0 0 0       % turn cursor on
```

commands

A command may be output for execution by the operating system by:

```
   OSCLI  send string to operating system              [] only
      string OSCLI
```

additional interfaces

The user may need to call operating system input/output routines that are not supported through Nucleus words, particularly for secondary devices such as the analogue ports and serial link. See the chapter 'Machine-code programming' for details.

music and sound event input

Modules provide music and sound input where required, through:

* input words – values are returned to the user program in the same way as with #IN and QKEY for QWERTY keyboard input

* music events – a music input device, such as music keyboard, generates events as a transparent side process of keyboard input – see the word ACT for general details.

synchronisation

Because sound output is via the sound event queue and referenced to the timebase, it is delayed relative to non-sound output, and input appears delayed relative to it. The user program may synchronise non-sound output to sound output by aligning it with real time, and synchronise the sound output to input, again by aligning it with real time. See the word QTIME for details.

9 Execution control

AMPLE provides powerful facilities to control the path of execution through the program:

* word definition – the word contents are executed wherever its name is used in the program
* conditional – the instruction sequence is or is not executed depending on the outcome of a decision
* definite loop – the sequence is repeated a definite number of times, determined in advance by a calculated value
* indefinite loop – the sequence is repeated until a certain outcome of a decision
* concurrent – the sequence is executed alongside other sequences, using the specified player.

The word definition is described in the chapter 'Programs and words'.

control structures

Each of AMPLE's execution control structures consists of two words used as a pair, and in some cases an optional word for use in between the main two. The first word of the pair ends with '(' and the second starts with ')', to make clear that they are complementary parts of a structure, for example:

```
... IF( ... )IF ...
```

Optional structure words have brackets at the start and end of their names, for example:

```
... IF( ... )ELSE( ... )IF ...
```

Each control structure word may only be used as part of the full structure, and this must be in a single word definition. For example, the following are not allowed:

```
"myif" [ IF( ]   ... myif ... )IF ...

"start" [ ... IF( ... ] "end" [ ... )IF ... )
... start ... end ...
```

Control structures may be nested:

```
   ... IF( ... FOR( ... )FOR ... )IF ...       % allowed
```

but they may not overlap:

```
   ... IF( ... FOR( ... )IF ... )FOR ...       % NOT allowed
```

All control structure words are '[] only', meaning that they can only be used in word definitions, and not in the command line.

conditionals and loops

Simple conditional execution is provided by the IF structure:

```
   IF(  start conditional sequence     [] only
      flag IF( ... )IF or flag IF( ... )ELSE( ... )IF
   )ELSE(  separate conditional sections       [] only
   )IF end conditional         [] only
```

A simple FOR loop structure handles definite loops:

```
   FOR( start definite loop    [] only
      countnumber FOR( ... )FOR
   )FOR  end definite loop     [] only
   INDEX  return loop index    [] only
      INDEX -> number
   COUNT  return loop count    [] only
      COUNT -> number
```

The REPEAT structure allows a wide variety of indeterminate loops:

```
   REP(  start indefinite loop [] only
   )REP  end indefinite loop   [] only
   )UNTIL(  exit from indefinite loop  [] only
```

condition expressions

The actions of IF(and)UNTIL(are controlled by a flag input value: ON or OFF. This is usually the result of an expression using one or more of the following operators:

```
   #<  test previous number is less than number
      number1 number2 #< -> flag (number1 < number2)
   #=  test numbers are equal
      number1 number2 #= -> flag (number1 = number2)
   #>  test previous number is greater than number
      number1 number2 #> -> flag (number1 > number2)
```

```
   SIGN  test number is negative
      number SIGN -> flag

   AND   AND bits of numbers
      number1 number2 AND -> ANDnumber (number1 AND number2)
   OR  OR bits of numbers
      number1 number2 OR -> ORnumber (number1 OR number2)
   XOR  exclusive-OR bits of numbers
      number1 number2 XOR -> number3

   NOT  invert sense of flag
      flag1 NOT -> flag2
```

Other number operators may also be used in the expression.

concurrency

AMPLE uses **players** to execute instruction sequences concurrently, that is, alongside each other as opposed to one after the other. In many respects, each player is like a separate AMPLE computer: it can run its own instruction sequence, has its own music environment, number stack and voices etc. However, all players have access to the same word definitions and global data, and can communicate with each other.

Player number 0 is special. It executes the system's command interpreter in an infinite loop, accepting and executing input from the user. Commands, editors and any user words executed by entering their names as commands are said to be run 'in player 0'. At this level of use, the system looks similar to other interactive micro-computer languages, like BASIC. In particular, the system is either waiting for command input OR executing a program, but never doing both at the same time.

Concurrency is achieved by using players 1 to 10. Each of these is idle until it receives an instruction sequence to carry out, and when it finishes this, it becomes idle again. Any player may issue a sequence to any player, including itself.

Players 1-10 are commonly used to run the parallel parts of a musical piece. The program often consists of 'part words' – the definitions of the musical parts – and a main word (usually called RUN) that simply gives each player its part to perform. RUN is entered as a command in player 0: it starts the players and then finishes, returning to the % prompt while the piece plays.

player-control instructions

The following words assign instruction sequences to players for concurrent execution:

```
   P(  start concurrent sequence       [] only
      playernumber P( ... )P
   )P  end concurrent sequence [] only
```

They are used along with:

```
   READY  get ready for players
   GO  start all players
```

Other words connected with the use of players are:

```
   IDLE  pass control to other players
   PNUM  leave player number
      PNUM -> number
```

stopping execution

AMPLE has a word to stop execution of all players and return control to the % prompt:

```
   STOP  stop program
```

AMPLE has no equivalent to BASIC's general GOTO statement.

10 Machine–code programming

AMPLE user programs can call operating system and user machine-code routines, and accomodate user routines in existing unused or specially-reserved memory.

calling routines

A single word allows an operating system or user machine-code routine, terminated by RTS, to be called from an AMPLE program, passing and receiving values via the processor registers:

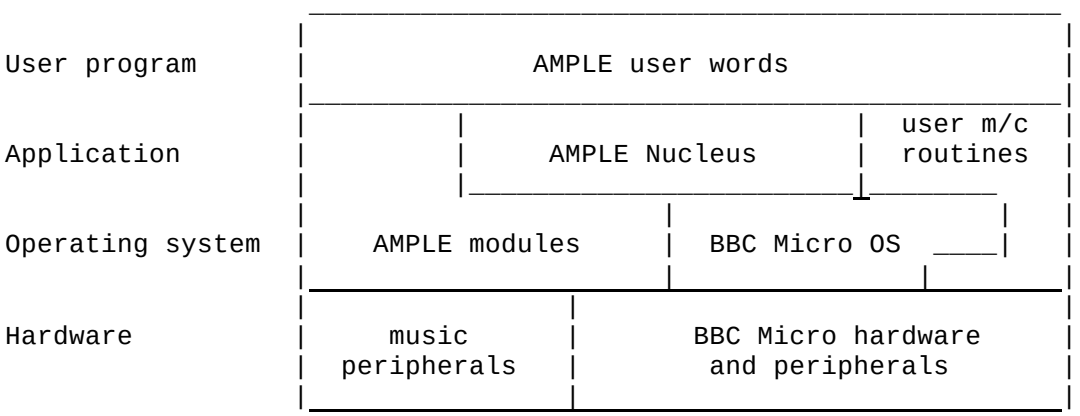
```
   CODE  call machine-code routine
      YXnumber CAnumber addressnumber CODE -> YXnumber PAnumber
```

User routines called in this way are free to call operating system routines as normal.

User machine-code routines may also be called directly by the operating system through the OS vector system, as normal. Routines should take care to follow the correct procedure so that AMPLE Nucleus and module vector intercepts are not disturbed. In particular, they should preserve all registers and exit by jumping to the vector's previous contents, that is, the address that was in the vector location before re-direction to the user routine.

user routine applications

The relationship of the various components of an AMPLE system, including user routines, can be illustrated as follows:



(Each component can use only the facilities of the components which are immediately below it.)

Note that user routines can be accessed by AMPLE user words and the operating system, and can access the operating system and BBC Micro hardware and peripherals. In particular, user routines cannot call any sort of AMPLE word or access the music hardware directly. This puts them on a level with the application and operating system (though they are physically part of the user program) and in fact the jobs for which user routines are most used can be thought of as extensions to AMPLE or the operating system.

Generally, there are two reasons for using a machine-code routine rather than an equivalent AMPLE definition:

* speed – where a computation needs to be carried out quickly, and a machine-code routine to do it is significantly faster than the AMPLE user word definition. Most AMPLE Nucleus words run at machine-code speed anyway, so there is little to be gained by replacing them or user definitions which use them intensively. The biggest speed increases will arise from replacing user word definitions which use many simple Nucleus words. The routine is accessed via a user word containing the CODE instruction, rather than a separate CODE at each use.

* vector interception – only machine-code routines can be called by operating system vectors.

Common applications include:

* computation – an AMPLE user word prepares the input data, calls the routine with CODE, and formats the output data. The complete user word is used as a formally-defined extension to the Nucleus.

* input – an additional hardware device can be handled by a machine-code routine that reports the results to the AMPLE user program. A simple method polls or scans the device using a machine-code routine (called by the user program) for speed, and another uses a vector intercept routine to poll at regular intervals, or respond to an interrupt, answering via a variable that is scanned by the user program. Any word that waits for an external event may use IDLE to avoid holding-up other players:

```
IDLE  pass control to other players
```

* output – high-speed output to new or existing devices can be provided, for example, for a graphics display using data computed and sent direct to the OS vdu drivers. User routines are called direct by the user program rather than by AMPLE's time control system, so their output is not automatically synchronised with normal music output. See the word QTIME for a method of synchronising user output.

user routines in language RAM

All language RAM from OSHWM (OSBYTE &83) to the bottom of display memory (OSBYTE &84) is reserved for use by the AMPLE system, so is not directly available for user routines. However, memory may be set aside using the DIM instruction, and machine code loaded into it using OSCLI, for example:

```
"codespace" [ 255 DIM ] % reserves 256 words, i.e. 512 bytes
...
READY               % clear all reserved memory
codespace           % reserve space
$&STR               % convert to string in hex
"LOAD code" $+      % make up command string: LOAD code addr
OSCLI               % execute load command
...
codespace           % later, find start address of code for use
...
```

It is important to remember that the address of the code space is completely variable, and may change on each run of the program. For this reason, the user machine code must either be location-independent, or be relocated to the load address each time it is loaded. Also, remember that READY clears all reserved memory, so the machine code must be reloaded after each use of READY.

user routines in operating system RAM

Operating system RAM that may be used for user routines includes unused primary workspace (for example, &BOO to &AFF under many conditions), and unused secondary workspace (between primary OSHWM (OSBYTE &B3) and current OSHWM (OSBYTE &83/&B4)).

The amount of secondary workspace may be increased to include the requirements of user routines by a user program that raises the value of secondary (current) OSHWM before the *AMPLE command of the AMPLE start-up sequence. (This reduces the amount of language RAM available to AMPLE accordingly.) A BBC BASIC program to do

this is as follows:

```
10 usersize=&200:REM amount required (a full number of pages)
20 osbyte=&FFF4:osbOSHWM=&B4
30 :
40 A%=osbOSHWM
50 Y%=&FF:X%=0
60 oshwm=USRosbyte
70 :
80 Y%=0:X%=(usersize+poshwm)DIV &100
90 dummy=USRosbyte
```

This program would be loaded and run immediately before the *AMPLE command in the system disc's standard start-up EXEC file. The program could be extended to set the amount of memory required to suit an existing user machine-code program, and then load it to the old value of OSHWM (variable 'oshwm'), relocating internal addresses if necessary. Alternatively, the program could assemble the user routines there and then, in place. It could also pass the (variable) entry addresses of the routines to the calling AMPLE program – see 'locating user routines' later.

An alternative method of allocating RAM is through use of a sideways ROM that claims private workspace through the normal mechanism on operating system start-up.

user routines in ROM

Routines in sideways ROM may be called from AMPLE, provided the standard operating system procedure for ROM selection is followed. This makes use of the ROM select register at &FE30, the ROM select register soft copy at &F4, a temporary location for storage of the AMPLE ROM number, and the number of the ROM containing the user routine. A typical BBC BASIC assembler program skeleton to this is as follows:

```
rsregister = &FE30          \ define symbols
rsregistercopy = &F4
...
.ampleromno EQUB 0          \ define workspace
.userromno EQUB 0
...
STA userromno               \ record user ROM number provided,
            \ for example, on ROM initialisation service call
...
.callroutine
\ select user rom
LDA rsregistercopy          \ get the current ROM number
STA ampleromno              \ save for later restoration
```

```
LDA userromno               \ get userromno
STA rsregistercopy          \ store in copy FIRST...
STA rsregister              \ then store in register
\ call user routine
JMP routine                 \ enter user routine, in ROM
\ reselect AMPLE ROM
LDA amplerom                \ get original rom number
STA rsregistercopy          \ store in copy FIRST...
STA rsregister              \ then store in register
\ return
RTS
```

locating user routines

The entry address of the user routine, whether in RAM or ROM, must be known to the AMPLE program for it to be called using CODE. Where the routine is not in AMPLE reserved memory, but in operating system RAM or sideways ROM, some means is needed to automatically pass the entry address to the program, since if it was included as an absolute literal number, it would have to be changed by hand to suit each new location of the user routine.

One method is to hold the routine entry address at a fixed address from which the AMPLE program reads it. For a RAM routine, this could be in primary operating system workspace and be written to by the code loader program. For a ROM routine, it could be at a fixed address near the start of the ROM, and the entry address written to it at assembly time. Where two or more entry addresses are needed, they can simply be stored at successive addresses.

user routines' zero page workspace

AMPLE leaves locations &8E and &8F free for use by user routines.

communication with user routines

User routines do not have access to any part of the AMPLE workspace or user program, so passing of values between AMPLE program and routine is accomplished by CODE's register value transferral and/or a special mechanism implemented by the user.

One such mechanism uses a block of memory to which both the program and routine have access. This could be either at a fixed address in operating system workspace, at a fixed offset from the routine entry address, or reserved by the AMPLE program with DIM. In the last case, the program passes the address to the routine in the Y and X registers each time it is called. The program and

routine agree the use of the locations at fixed offsets from the start of the block.

11 Errors

An error is a condition arising when the system detects that it is unable to do what user input demands of it. This is usually caused by a fault in a program or command. Errors may be detected by AMPLE or the operating system (including the filing system, etc.), but are treated in the same way whatever the source.

error effects

All errors give an **error message**, an explanatory single-line message beginning with !. There are three types:

description	example
simple	! Mistake
with location	! No number in tune
with location and player	! Bad bar in page1 in player 1

Individual error messages are explained later on.

Depending on how and where the error occurred, its effect may include:

* returning to command mode, if the user was in an edit mode
* stopping execution of the command line, so remaining commands are ignored
* stopping execution of the word executed at the command line, and returning to the % prompt
* if it occured in the definition of a word (inside [...]), printing an additional '!' to indicate the site of an error, and aborting the definition.
* stopping execution of players 1-10
* silencing all sounds
* executing a SCORE in player 0 if a chord or key signature was in progress in player 0. This ends the chord or key signature, and restores the music environment to a defined state.

In particular, the effects of each class of error are as follows:

* in the command line, or in an AMPLE word in the command line:
  * a simple error message is printed
  * the rest of the command line is ignored.

* in the command line, while defining a word (inside []):
  * a '!' is printed under the error site
  * a simple error message is printed
  * the definition is aborted and the rest of the line ignored

* in a user word in player 0:
  * an error message with location is printed
  * execution in player 0 stops and returns to the %
  * the rest of the command line is ignored
  * SCORE is executed if necessary.

* in one of players 1-10:
  * an error message with location and player is printed
  * execution of player 0 stops and returns to the %
  * the rest of the command line is ignored
  * SCORE is executed if necessary
  * players 1-10 are stopped
  * all sounds are silenced
  * HALT and PAUSE states are set to 'off'.

  All effects apart from the message printing are exactly as if STOP had been executed.

  An ESCAPE key press has the same effect as an error in one of players 1-10, but a simple error message is printed, without location or player number.

error effects from modules

Each voice-providing module is responsible for silencing its own voices on those error types that silence all sounds. How this is carried out depends on the module, but ideally it should give complete and immediate silence with no further disturbance to the voices, that is, no alteration to voice control values. Some modules may be unable to achieve silence, for example, if they are driving remote physical voices through a simple interface. In this case, they use the best available alternative, such as sending an 'off gate' message to each voice, and letting the sounds end naturally as if in rests.

errors and editors

Each editor may provide extra error effects and reporting facilities specific to its function, in particular, for locating the site of an error in the data being edited. Three types of error location are common:

* of the site of an error given when the data was executed with the editor's 'direct execute' facility, if it has one

* of the site of an error given when the data was made into a word – an alternative to the % prompt's '!' indicator

* of the site of the last error to be reported, when the word containing it is subsequently called into the editor.

Often, the editor indicates the location by automatically positioning the edit mode cursor on it on entering edit mode.

error-like events

Certain events which are not errors have some of the same side effects as some errors, including:

* SCORE is executed if necessary
* players 1-10 are stopped
* all sounds are silenced
* HALT and PAUSE states are set to 'off'.

and the following additional effects:

* the music action list of player 0 is cleared
* the record of the location of the last error in a user word, available to editors for special locating, is cleared

These events arise from any command that re-arranges memory and therefore interrapts the execution of the user program. The commands in this catagory are:

```
NEW, LOAD, SAVE
COMPACT
MLOAD, INSTALL, MDELETE
DELETE, RENAME,
[ ... ] (when redefining an existing word)
```

Note that MODE is not one of these, so it is usable from within the user program.

error messages

AMPLE's error messages are as follows, in alphabetical order. The operating system and its services (including the filing system) have their own error messages, and you should consult their documentation if you get any message not listed below.

number
A serious fault has arisen in the system, probably as a result of memory being corrupted.

If this occurs, you should restart the language from the system disc (or with *AMPLE). You can save the program first, but in extreme cases this may also have been corrupted and the file will be rejected by LOAD.

A faulty program can corrupt memory by incorrect use of a store operator (#! or #B!), for example if when assigning a value to a variable, the name of the variable word is missing.

One numbered error that cannot be cured by re-starting the system is error 8. This rneans 'Too many servers', meaning that an attempt was made to install too many voice-providing modules. This can only arise from an incorrect installation sequence on a user-created system disc. The maximum number of voice-providing modules is seven.

Already present
An attempt was made to MLOAD or INSTALL a module that was already in memory. The MLOAD or INSTALL does not take place.

Bad bar
The total length of a bar did not match the bar length set by BAR. This is issued by '|' and usually arises from an extra or missing note in a bar, or a missing barline.

If you are entering a sequence of AMPLE notation at the % prompt by copying from a listing on the screen, you can avoid errors from incomplete bars by entering '0 BAR' first.

Bad context
A word has been used in an incorrect context, for example, using a command such as LOAD inside a word definition, or a word that is only allowed inside word definitions, like $2, as a command.

The following words and types of word can cause this error:

word	fault
'command only' word	inside [...]
'[] only' word	as a command, that is, not inside []
)	no (
)K	no K(
(...)	inside K( )K or ( )
K(...)K	inside K( )K or ( )
GO	inside one of players 1-10
READY	inside one of players 1-10
INDEX or COUNT	not directly inside FOR(...)FOR

Bad element
An attempt was made to access an array element outside the range dimensioned. This error is issued by ARRAY if it finds the element number to be less than 0 or greater than the number dimensioned in the preceeding DIM. The error message gives the name of the array word, not of the word in which it was used.

This error is also issued by FVAR when an attempt is made to access stack items that are beyond the range of the stack, for example if 4 FVAR is executed when there are only 3 values on the stack below the FRAME point, and when the stack pointer is found to be below the FRAME point.

Bad hex
The first character after & was not a valid hex digit. This error is given by & and &VAL.

Bad load address
An attempt was made to load a module which requires a particular load address, and memory space was not available at this address. Normal modules can be loaded at any address, and only very special modules are able to cause this error.

A module may independently require a load address to be a certain range, for example outside the region of shadow RAM, and will therefore issue this error itself under certain conditions. These conditions should not arise under normal use of the module.

Bad mode
There was not enough free memory for the mode requested by MODE or a module.

If this occurs from MODE typed in directly as a command or issued from a user program or, you should enter COMPACT and try again. COMPACT is done automatically on LOAD and SAVE.

Remember that loaded modules take up memory and loading another module may stop you switching to a particular mode with a par ticular size program.

On BBC Microcomputers with shadow RAM, this error does not appear.

Bad module
The file loaded with INSTALL or MLOAD was not a valid module. This happens if you try to MLOAD an AMPLE program, for example.

Bad name
There was a fault in the name given for a new word, either in '[' or RENAME. It was probably too long (longer than 15 characters) or null ("").

Bad player number
An invalid player number was given. The range for P(is 1 to 10, and for SHARE is 0 to 10.

Bad program
The LOADed file was not an AMPLE Nucleus program produced by SAVE. It is possible to get the 'Too big' message for the same reason. This error leaves a clear program, as if you had used NEW.

Bad ROM
The Nucleus ROM image is faulty, due to a fault in computer or the ROM IC itself. Before suspecting the ROM, you should try it on another computer. The ROM image is checked on *AMPLE (start-up from the system disc), AMPLE and BREAK inside AMPLE.

Bad string
There was no closing single " in the string. Strings may not stretch from one line to the next.

Bad structure
A structure mismatch was found. In particular, a structure end or middle word did not match the last unmatched structure beginning or middle word, for example

```
FOR( ... IF( ... ) FOR
```

Each word beginning with) should be used only after its matching word ending with (. The words that can issue this error are:

```
)FOR, )REP, )UNTIL(, )IF, )ELSE(, )P, ]
```

Note that though)K and) are each part of a pair, they are music environment words and not true structure words, so they give the 'Bad context' error when mismatched.

Division by zero
An attempt was made to divide by zero. This usually occurs from #/, but other system words that carry out division can also issue it. For example, +T, -T, +L and -L issue this error if the ',' setting is 0, (inside a chord, for example).

Escape
The ESCAPE key was pressed. ESCAPE stops all players and sounds, and returns control to the keyboard.

Extra number
There was a number left on the stack after the input line had been executed.

This arises when there is a surplus number on the line or in a word on the line, that is, a number which is not used by following words. For example,

```
24,bDc 12DE 24,G    (missing comma after 12)
1 00 OFFSET         (space in number 100)
```

In complicated programs where the stack is used for temporary storage, this error can result from faulty program structure or

logic. Note that if such a fault arises in one of players 1-10, it will not be detected unless it is repeated so as to cause a 'Too many numbers' error.

Extra string
There was a string left on the stack after the input line had been executed.

This usually means that you gave a string argument where one was not required, for example:

```
 "tune" WRITE
```

Care must be taken when using two double quotes to include a double quote in a string.

Fixed module
An attempt was made to delete a fixed module (status F) with MDELETE.

Only modules that are temporary (status T) or program-loaded (status P) may be deleted. You can find out the status of a loaded module with MCAT.

In use
An attempt was made to DELETE a word, or MDELETE a module, that was in use. You can use FIND to locate any usages in the program. The module could alternatively be in use as the current editor. MCAT indicates both kinds of module use.

To delete a recursive word, you should first redefine it to an empty definition, for example, "fact" []

Mistake
Characters on the input line were not understood, i.e. were not recognised as a word, number or string.

If you mis-type a command, the system will interpret as much of it as it can in terms of the existing user and system words before it is forced to give the 'Mistake' message.

This error can sometimes arise if an essential separating space is left out, so two names are run together and unintentionally produce the name of another existing word. For example, if

```
   part1 act         was entered as       part1act
```

and there was a word called 'part1a', this longer word would be recognised. If there was no word called 'ct', the 'Mistake' message would appear.

No number
A number was missing i.e. the number stack was empty when a word attempted to remove a number. This is usually the result of leaving an argument out, for example

```
 MODE       (should be, for example, 7 MODE)
```

In complicated programs that use the stack for temporary storage, this often results from a programming fault. If there was a temporary number on the stack when a word attempted to use the missing number on top, the temporary number will be used instead, so you should be careful to test the individual sections of complicated words, preferably by defining them as words.

No room
There was not enough free memory for the operation. This error can be produced in a word definition, during editing, and by P(, ACT(and DIM.

There may be enough free memory in total, but split up so that the largest single piece is too small. COMPACT arranges all the free memory into one piece.

There may still be players in existence from the last run of a program. The space they consume can be recovered by discarding them with ESCAPE or STOP. COMPACT does this automatically.

No string
A string was missing i.e. the string stack was empty when a word attempted to remove a string.

You may have left out an argument, or put it in the wrong place, for example

```
   SAVE "temp"
```

The keyboard input line is always on the string stack, so that inside a word, this will be used instead of a missing string and the error will appear when the current directly-executed word finishes.

No such item
There is no word or module of the given name.

Note that the case of word and module names is significant, so this error can result from you entering the name in the wrong case.

Too big
There is not enough free memory to load the program or module.

Memory is shared between the program, module, screen (on non-shadow memory computers) and the BBC Micro operating system. If you change the amount of memory used by any one of these, you could end up with too little for a program or module that previously loaded successfully. Suggested remedies are as follows:

- reduce the program size
- remove unwanted modules (or use a smaller installation)
- change to a more economical screen mode, for example, mode 7
- remove or disable ROMs that increase operating system memory usage

This error may also arise if the file loaded is unrecognisable as a valid AMPLE program, where one would have expected the 'Bad program' error.

Too many characters
The maximum total length of strings on the string stack was exceeded. It can hold 128 characters.

Too many levels
The capacity of the player's return stack was exceeded. This can happen in the following cases:

1. a too-deep nesting of words was used
2. a too-deep nesting of FOR loops was used
3. a too-long chain of music actions was used

Too many numbers
The capacity of the number stack was exceeded. It can hold 32 numbers.

The commonest cause of this error is a loop that leaves an extra number on the stack each time around.

Too many strings
The capacity of the string stack was exceeded. It can hold 16 strings.

Too many modules
A limit on the number of loaded modules was exceeded. This limit applies only to modules that provide words for use in user word definition, so, for example, most editor modules will not contribute to this. The maximum number allowed is 9.

Too many voices
All voices of a particular type were already in use when an attempt was made to assign another. Voices are provided by modules, each of which has a limit on the number of voices. See the appropriate application User Guide for information.

Too many words
The maximum number of user words allowed had already been reached. The maximum number of words allowed is 125.

Warning: duplicate name
The new name specified to RENAME was already in use for another user word. This message is a warning only, and the operation is still carried out. All subsequent references to the name will refer to the new word. You can use RENAME again to change the name.

Part 2 - Reference

12 Dictionary of words

This chapter provides a detailed description of every AMPLE Nucleus word and symbol, for reference purposes. An initial index gives word names and basic information, and the main dictionary has a full entry for each word and symbol.

The entries are arranged by name in a lexicographic (dictionary-type) order based on the following order of characters:

```
! " # $ % & ' ( ) * + , - / : ; < = > @ [ \ ] ^ | ~  0-9  A-Z
```

Any non-letters at the start of the name are counted as if they were at the end, that is, they only affect the ordering of otherwise-identical names. For example, &VAL appears after VAL even though & appears before VAL. This means you can roughly locate any name by its letters, ignoring any non-letters in it.

The general form of an entry is as follows:

```
word name  function                                    status
input items -> output items

description

example(s)

related words

further information
```

word name

The word name is shown exactly as you type it in, except in some cases where a description of the name in angle brackets is given instead. For example, <carriage return> means the carriage return character.

function and status

A short description of the word's function is given after the name, sometimes followed by a status item which is one of the following:

command	The word can only be used as a command entered directly at the keyboard. It cannot be used in word definitions, that is, between [and].
[] only	The word can only be used between [and], that is, in word definitions. It cannot be typed in directly as a command.

Where there is no status indication, the word can be used both as a command and in word definitions.

input and output items

Where the word takes input items (arguments) and/or delivers output items (results), these are indicated on the next line. Each item (number1, for example) is explained in the text. Input items are shown before the word, just as you supply them when using it, for example:

```
indication:    number MODE
example use:   7 MODE
```

Some words deliver output items. Their input/output descriptions have a -> sign followed by the list of output items which the word delivers. These are shown exactly as if you had entered them in place of the word and its inputs, for example:

```
indication:    number1 number2 #+ -> number3
example use:   1 2 #+   produces   3
```

Some words accept and/or deliver numbers (or logical flags) and strings. Since strings and numbers are held independently on separate stacks, the position of a string relative to that of a number, and vice versa, is immaterial. For example,

```
indication:         string number $- -> rightstring leftstring
example use:        "hello" 1 $-
equivalent use:     1 "hello" $-
```

A few words are special symbols that go before a group of characters, rather than taking a string as an input item. An example of this is '*' which treats the rest of the line as an operating system command. This type of input item is shown as a

description of characters in angle brackets (< >), for example, the 'input/output items' line for '*' is:

```
*<line>
```

description

The description of the word gives all the essential information in a concise form. More general information on the subject of the word and similar words can often be found in other chapters - use the index to find these references.

examples

There are one or more short examples for all but the very simplest words. These are not intended to be fully-functional programs for typing-in, but concise extracts showing the use and function of the word. In particular, some of them use '[] only' words so they can only be used in a word definition. For the most important words that are used in programs there are complete example word definitions in other chapters.

related words

Next, there is a list of related words. This includes words which are often used with the word being described, and others which have related or alternative functions in which you might be interested.

further information

This section of the entry only appears for some words and has additional information for advanced users only, often with further examples.

index of words

```
&  indicate hexadecimal number
   &<hex digits> -> number
'  accent next note or hit
(  start additional chord notes
)  end additional chord notes
*  indicate operating system command                    command
   *<line>
+  sharpen next note
,  set length
      number ,
-  flatten next note or indicate negative number
/  hold music event
:  set octave
      number:
;  set music voice
      voicenumber ;
=  naturalise next note
@  set transposition in semitones
      transnumber @
[  start word definition                                command
      namestring [...]
\  move back
]  end word definition
~  slur next note                                       [] only
0 to 9  decimal digits
   <decimal digits> -> number
^  play rest
^;  play chord rest
|  mark end of bar
A to G  play note with ascending pitch
a to g  play note with descending pitch
ACT  execute music action
   see text
ACT(  start music action sequence                       [] only
   positionnumber ACT(
)ACT  end music action sequence                         [] only
ALIGN  ensure text cursor is at start of line
AMPLE  restart system                                   command
AND  AND bits of numbers
   number1 number2 AND -> ANDnumber
ARRAY  access array element
   elementnumber baseaddressnumber ARRAY -> addressnumber
ASC  convert character to number
   string ASC -> asciinumber                            [] only
#B!  store low byte of number at address
   datanumber addressnumber #B!
#B12  swap high and low bytes of number
   number2 #B12 -> number2
```

```
#B?  fetch byte from address
   addressnumber #B? -> datanumber
BAR  set bar length in length units
   lengthsnumber BAR
$CHR  convert number to character                    [] only
   asciinumber $CHR -> string
CLEAR  clear editor data                             command
CODE  call machine-code routine
   YXnumber CAnumber addressnumber CODE -> YXnumber PAnumber
COMPACT  compact unused memory                       command
COUNT  return loop count                             [] only
   COUNT -> number
DELETE  delete word                                  command
   namestring DELETE
DIM  reserve memory
   sizenumber DIM -> addressnumber
DISPLAY  display text
DURATION  wait for a period of time
   number DURATION
)ELSE(  separate conditional sections                [] only
EVERY  leave 'every' value
   EVERY -> number
FAST  select fast/normal tempo
   flag FAST
FCOPY  copy numbers from frame pointer
   number -> number1 ... number-n
FIND  find uses of word                              command
   namestring FIND
FOR(  start definite loop                            [] only
   countnumber FOR( .... )FOR
)FOR  end definite loop                              [] only
FRAME  set frame pointer to top of stack
FRAME!  write frame pointer
   pointernumber FRAME!
FRAME?  read frame pointer
   FRAME? -> pointernumber
FVAR  access stack frame item
   elementnumber FVAR -> addressnumber
GO  start players together
GVAR  create variable                                [] only
   GVAR -> addressnumber
HALT  halt/continue timebase
   flag HALT
#IN  wait for and get keypress
   #IN -> asciinumber
$IN  input line from keyboard                        [] only
   $IN -> string
IDLE  pass control to other players
IF(  start conditional sequence                      [] only
   flag IF( ... )IF  or  flag IF( ... )ELSE( ... )IF
```

```
)IF  end conditional                                        [] only
INDEX  leave loop index                                     [] only
   INDEX -> number
INSTALL  install module                                     command
   namestring INSTALL
K(  start key signature
)K  end key signature
LEN  get length of string                                   [] only
   string LEN -> string lengthnumber
LOAD  load program                                          command
   namestring LOAD
'L  set accent strength
   number 'L
=L  set dynamic level
   number =L
+L  increase dynamic level
   changenumber eventsnumber +L
-L  decrease dynamic level
   changenumber eventsnumber -L
MAX  leave larger of two numbers
   number1 number2 -> largestnumber
MCAT  display catalogue of modules                          command
MDELETE  delete module                                      command
   namestring MDELETE
MEM  show memory usage in bytes                             command
MIN  leave smaller of two numbers
   number1 number2 -> smallestnumber
MLOAD  load module                                          command
   namestring MLOAD
MODE  enter display mode                                    command
   number MODE
MPREFIX  set module filename prefix                         command
   string MPREFIX
MSHOW  show list of words in module                         command
   modnamestring MSHOW
MVAL!  write music variables
   framelev keysig barcountlen octnote length tranvoice MVAL!
MVAL?  read music variables
   MVAL? -> framelev keysig barcountlen octnote length tranvoice
NEW  discard program                                        command
NL  print new line
NOT  invert sense of flag
   flag1 NOT -> flag2
NOUT  print number in decimal
   number NOUT
&NOUT  print number in hexadecimal
   number &NOUT
OFF  leave off flag value
   OFF -> offflag
```

```
ON  leave on flag value
   ON -> onflag
OR  OR bits of numbers
   number1 number2 OR -> ORnumber
OSCLI  send string to operating system                [] only
   string OSCLI
#OUT  send ASCII code to screen
   number #OUT
$OUT  print string                                    [] only
   string $OUT
$PAD  pad string to length with spaces                [] only
   string1 lengthnumber $PAD -> string2
P(  start concurrent sequence                         [] only
   playernumber P( ... )P
)P  end concurrent sequence                           [] only
PAUSE  pause/continue sound processing
   flag PAUSE
PNUM  leave player number
   PNUM -> number
QKEY  test key status or get keypress
   negativenumber QKEY -> flag
   zeronumber QKEY -> asciinumber
QTIME  return queue time
   QTIME -> number
QUIT  leave editor                                    command
RAND  get random number
   RAND -> number
RAND!  set starting point for random numbers
   number RAND!
RANDL  get random number in range
   maxnumber RANDL -> number
READY  make system ready
RENAME  rename word                                   command
   oldnamestring newnamestring RENAME
REP(  start indefinite loop                           [] only
)REP  end indefinite loop                             [] only
$REV  reverse the order of characters                 [] only
   string $REV -> reversedstring
RVOICES  set voices range
   startnumber endnumber RVOICES
SAVE  save program                                    command
   string SAVE
SCORE  prepare for music words
SHARE  select voice ensemble
   ensemblenumber SHARE
SHOW  show user words                                 command
SIGN  test number is negative
   number SIGN -> flag
SIMPLEACT  remove all music actions
SP  print a space
```

```
STOP  stop program
$STR  convert number to decimal string representation  [] only
   number $STR -> string
&$STR  convert number to hex string representation     [] only
   number &$STR -> string
$STRIP  remove leading spaces from string              [] only
   string1 $STRIP -> string2
=T  set tempo
   number =T
+T  increase tempo
   changenumber beatsnumber +T
-T  decrease tempo
   changenumber beatsnumber -T
TYPE  type word definition on the screen               command
   namestring TYPE
)UNTIL(  exit from indefinite loop                     [] only
UNUSED  make voice(s) unused
VAL  convert string to unsigned decimal number         [] only
   string VAL -> remainingstring number ON    if found
   string VAL -> remainingstring OFF          if not found
&VAL  convert to unsigned hex number                   [] only
   string &VAL -> remainingstring number ON   if found
   string &VAL -> remainingstring OFF         if not found
VOICE  select voice(s)
   voicenumber VOICE
VOICE!  set voice settings in frame
   voicenumber VOICE!   or   EVERY VOICE!
VOICES  set number of voices
   number VOICES
WIND  advance time
   ticksnumber WIND
WRITE  display text of all words                       command
X  play hit
XOR  exclusive-OR bits of numbers
   number1 number2 XOR -> XORnumber
```

<cr> mark line end

The system includes a word whose name is just a single carriage return character {code 13), for the purpose of representing line ends inside words. When a word definition is entered, each line of text, except the last line, ends with a carriage return character. The carriage return word is included at this point in the same way as any word with a normal name, thereby marking the end of the line. When the word definition is displayed, with TYPE for example, the effect of the carriage return is to start a new line on the display.

The carriage return also marks the end of comments, so that after %, only the characters up to the end of that line are ignored, and characters on the next line are treated as normal.

related words NL ALIGN

further information

A literal (quoted) string cannot contain a carriage return, since it cannot stretch from one line to the next, and there is no special mechanism for including control codes in literal strings. However, you can include a carriage return in a string on the stack by first creating a one-character string containing the code using $CHR, for example:

```
13 $CHR             % create string with just CR
"HELLO" $+          % make "HELLO<CR>"
```

To print a carriage return, use NL:

```
"HELLO" $OUT NL     % print HELLO followed by CR
```

Carriage returns cannot be included in comments.

<space> separate items

The space character is dealt with by the system as a word much like words with normal names. It does nothing when executed, but is important for separating items and making words more readable. It is stored in word definitions so that spacing is preserved as expected.

A space must be put between two items that could be misinterpreted (by the user or the system) if they were run together, for example numbers, strings, and some words. In any event, you should

separate all words, except in lines of music words where this would waste a lot of space.

examples

```
1 2 #+ NOUT          % space needed between numbers

"keyl" "key2"NAME    % space needed between strings to
                     % avoid "" in longer string

1 VOICE Simpleins    % space between VOICE and Simpleins is
                     % essential to avoid VOICES
```

further information

The system stores spaces in compacted form so that a group of more than two spaces uses no more storage space than just two spaces. A single space uses one byte and two or more spaces use two bytes.

! move an extra octave up or down

The '!' ('pling') word moves the pitch of the next note by one extra octave, that is, if it is upper case, it raises it an octave, and if it is lower case, it lowers it an octave. This applies even if it is in between two notes of the same case.

You use it for pitch jumps of more than an octave, and jumps of an octave between notes of the same case. '!' is entirely equivalent to the appropriate ':' setting at the same point, but '!' is relative to the last note so is more suited for use in the middle of a line or phrase, whereas ':' is intended for the start of the line or phrase.

Any number of '!' signs can be put before a note to jump by two or more extra octaves.

examples

```
1:c!C              is equivalent to       1:c 3:c

1:c!c              is equivalent to       1:c 0:c

1:C!c              is equivalent to       1:C 2:C

1:C!!C             is equivalent to       l:C 3:C

1:!C               is equivalent to       2:C
```

"
start literal string

```
"characters" -> string                    inside []
string1 "characters" -> string2 string1   outside [ ... ]
```

The double quote character is used to include strings of characters in a program or command. Double quotes are put at the start and end of the string of characters. To include the double quote character itself in the string, two double quotes are included. The string cannot stretch over the end of the line.

examples

```
"program"SAVE               % filename provided as a string

"HELLO" $OUT                % print   HELLO

"Say ""HELLO""" $OUT        % print   Say "HELLO"
```

further information

Control codes cannot be included in literal (quoted) strings, but a string containing a control code can be made on the stack with $CHR, and this can then be added to a literal string using $+, for example:

```
                     % to include CR in a string
13 $CHR              % create string with just CR
"HELLO" $+           % make "HELLO<CR>"
```

Strings can contain any 8-bit code, included in this way.

In direct mode, that is, outside [...], strings operate differently because the system uses the string stack to hold the input line (command line) as a string at all times. Normally, you only use the string stack inside word definitions, that is, any word executed from the command line must preserve the state of the string stack.

Whereas inside [...], "..." puts the string on the top of the stack, when entered in the command line, it puts the string as the second item down, that is, underneath the input line string. So, the complete stack action of " is actually:

```
"characters" -> string                             inside  []
inputstring "characters" -> string2 inputstring    outside []
```

Unlike the number stack, there is only one string stack, accessed equally by all players. When player 0 is using the string stack, for the command interpreter for example, all other players should

preserve the state of the stack over each IDLE or other possible idle (see IDLE for details of possible idles).

#! store number at address

```
datanumber addressnumber #!
```

#! stores number at a particular address. It is mostly used for storing values in variables and arrays, and usually appears immediately after the variable name.

examples

```
"total" [GVAR] ...            % variable 'total' ...
0 total #!                    % set to zero

"totals" [ 10 DIM ARRAY ] ...  % array 'totals' ...
1  0 totals #!                % element 0 set to 1

xvar #? yvar #!     % transfer value from xvar to yvar
```

#! will operate on any address and should therefore be used with care to avoid corrupting meMory.

related words #? #+! #B! #B? GVAR DIM ARRAY

further information

#! stores the low byte of the number at the address, and the high byte at the address plus one.

#* multiply two numbers

```
number1 number2 #* -> productnumber (number1 x number2)
```

#* multiplies the two numbers together, leaving the product.

example -2 3 #* produces -6

related words #+ #- #/

#+ add two numbers

```
number1 number2 #+ -> sumnumber (number1 + number2)
```

#+ adds the two numbers together, leaving the sum.

examples

```
2 3 #+   produces   5
```

```
... #11 #+ ...     % fast multiply by two
```

related words #- #* #/

#+! add number to number at address

```
datanumber addressnumber #+!
```

#+! adds the number to the number at the address given. It is used for adding numbers to variables and array elements and in most cases it is put immediately after the variable or array name.

It provides a more efficient alternative to fetching, adding and storing:

```
1 count #+!   is equivalent to   count #? 1 #+ count #!
```

#+! will operate on any address and should therefore be used with care to avoid corrupting memory.

example

```
1 total #+!        % add one to total
-1 total #+!       % subtract one from total
```

related words #! #? #B! #B? GVAR DIM ARRAY

#- subtract number from previous number

```
number1 number2 #- -> differencenumber (number1 - number2)
```

#- subtracts the number from the one before it, leaving the difference.

examples

```
5 1 #-       produces   4
5 1 #12 #-   produces  -4

"neg" [ % change sign
0 #12 #- ]
...
2 neg        produces  -2
```

related words #+ #* #/

#/ divide previous number by number

```
number1 number2 #/ -> quotientnumber remaindernumber
```

#/ divides the number into the one before it, leaving both the quotient (integer part) and the remainder (fractional part).

For a simple integer division (like BASIC DIV), the remainder is discarded using f2:

```
   number1 number2 #/ #2 -> quotientnumber (number1 DIV number2)
```

If only the remainder is needed (the result of BASIC MOD), the quotient is discarded:

```
   number1 number2 #/ #12 #2 -> remaindernumber
                                (number1 MOD number2)
```

examples

```
10 3 #/          produces   3 1   % full result
10 3 #/ #2       produces   3     % quotient only
10 3 #/ #12 #2   produces   1     % remainder only

"mod" [ % integer remainder
#/ #12 #2 ]
...
10 3 mod         produces   1
```

related words #+ #- #*

further information

For each combination of argument signs, the results signs are as follows:

number1 (dividend)	number2 (divisor)		quotient	remainder
+	+	#/ ->	+	+
-	+	#/ ->	-	-
+	-	#/ ->	-	+
-	-	#/ ->	+	-

#11 duplicate number

```
   number fl1 -> number number
```

#11 makes a copy of the number, that is, it leaves two numbers instead of the one. It is used when there are two operations required on the same number, for example, printing out, and storing in a variable.

examples

```
4 #11 produces     4 4

... #11 NOUT count #!      % print and store

#11 0 #= IF( ... )IF       % non-destructive test
```

related words #2 #l2 #212 #2121 #213

#12 swap two numbers

number2 number1 #12 -> number1 number2

#12 exchanges the top two items on the number stack. It allows the first-supplied of two numbers to be accessed while keeping the other for a further operation. It is often used to exchange two input items that have been supplied in reverse order for convenience.

examples
```
8 5 #12  produces   5 8
... 0 #12 #- ...           % negate
"mod" [ #/ #12 #2 ]        % num1 num2 -> remainder
```

related words #2 #11 #212 #2121 #213

#2 discard number

number #2

#2 discards the number. This is often used to discard an unwanted product of an operation or the number left after a list of operations that duplicate it.

examples
```
2 1 02     produces   2
5 2 #/ #2                  % remainder discarded

#IN                        % number supplied ...
10 FOR(
#11 ...                    % ... repeatedly used ...
)FOR #2                    % .. and then discarded
```

related words #11 #12 #212 #2121 #213

#212 duplicate previous number

number2 number1 #212 -> number2 number1 number2

#212 copies the second-from-top number to the top of the stack, leaving three numbers instead of two. It allows easier access when two or more numbers are being operated on:

```
                    ... #212 NOUT ...
is equivalent to    ... #12 #11 NOUT #12 ...
```

example 3 2 1 #212 produces 3 2 1 2

related words #2 #11 #12 #2121 #213

#2121 duplicate number and previous number

number2 number1 #2121 -> number2 number1 number2 number1

#2121 makes copies of two numbers, leaving four instead of two. It is useful for performing an operation on two numbers while leaving them for a further operation.

examples

```
4 7 #2121   produces   4 7 4 7
#2121 #= IF( ... )IF       % non-destructive test
```

related words #2 #11 #12 #212 #2121 #213

#213 rotate positions of three numbers

number3 number2 number1 #213 -> number2 number1 number3

#213 moves the three numbers so that the third-from-top number is moved to the top of the stack, and the other two are moved down accordingly. It allows access to the third number on the stack, so that with careful planning, three temporary values can be stored during calculations.

examples

```
3 2 1 #213   produces   2 1 3

"#132" [ #213 #213 ]
...
3 2 1 #132   produces   1 3 2
```

related words #2 #11 #12 #212 #2121

#< test previous number is less than number

number1 number2 #< -> flag (number1 < number2)

#< compares the two numbers and leaves ON if number1 is less than number2, and OFF otherwise, removing the numbers. Remember that the order of numbers is such that 'number #<' asks 'is it less than this number'.

examples

```
4 0 #<   produces   OFF
4 6 #<   produces   ON

... 0 #> IF( ... )IF       % do if positive and not 0

"#>=" [ #< NOT ]           % 'greater than or equal'
```

related words #> #=

further information

#< works on signed values, so take care when using it on 16-bit unsigned values such as addresses, if they could be greater than &7FFF.

#= test numbers are equal

number1 number2 #= -> flag (number1 = number2)

#= compares the two numbers and leaves ON if the top two numbers were equal, and OFF otherwise, removing the numbers.

examples

```
4 4 #=   produces   ON
4 5 #=   produces   OFF

... 0 #= IF( ... )IF       % do if equal to 0

"#<>" [ #= NOT ]           % 'not equal to' ]
```

related words #< #>

#> test previous number is greater than number

number1 number2 #> -> flag (number1 > number2)

#> compares the two numbers and leaves ON if number1 is greater than number2, and OFF otherwise, removing the numbers. Remember that the order of numbers is such that 'number #>' asks 'is it greater than this number'.

```
Examples    -2 0 #>   produces   OFF
             3 2 #>   produces   ON

            ... 0 #> IF( ... )IF       % do if greater than zero

            "#<=" [ #> NOT ]           % 'less than or equal'
```

related words #< #=

further information

#< works on signed values, so take care when using it on 16-bit unsigned values such as addresses, if they could be greater than &7FFF.

#? fetch number from address

addressnumber #? -> datanumber

#? fetches the number from the address given. It is mainly used for reading variables and array elements, and is put immediately after the variable or array name.

examples

```
total f? NOUT      % print variable 'total'
3 totals #? NOUT   % print element 3 of array 'totals'
```

related words #! #+! #B! #B? GVAR DIM ARRAY

further information

The low byte is at the address, and the high byte is at the address plus one.

$+ add string to left end of previous string [] only

rightstring leftstring $+ -> string (leftstring + rightstring)

$+ adds (concatenates) the two strings, with the second (top) string going on the beginning (left end) of the first-supplied one. To add the strings the other way around, use $12 to exchange them first.

examples

"there" "hello" $+	produces	"hello there"
"hello" "there" $12 $+	produces	"hello there"
13 $CHR "hello" $+	produces	"hello<CR>"
"hello" "" $+	produces	"hello"

related words $- $2 $12 $REV $CHR

$- split string after numbered character [] only

string number $- -> rightstring leftstring

$- splits the string after the numbered character position, leaving the left part with the remaining right part underneath. With further words, it can be used to extract any left, middle or right part of the string.

examples

"hello" 2 $-	produces	"llo" "he"

```
"$rest" [ % string lennumber $left -> restofstring
$- $2 ]
...
"Item 1" 5 $rest    produces     "1"

"$mid" [ % string lennumber startnumber -> midstring
$- $2 $- $12 $2 ]
...
"hello" 2 1 $mid    produces     "el"
```

related words $+ $2 $12 $REV $CHR

further information

Either or both results can be null strings. If the split position is less than zero or greater than the string length, then the split is made at the nearest limit.

$12 swap two strings [] only

```
string1 string2 $12 -> string2 string1
```

$12 exchanges the two strings. A common use is before $+ to add the strings in the opposite order.

examples

```
"hello" "there" $12   produces   "there" "hello"

"2" " items" $12 $+   produces   "2 items"
```

related words $+ f- $2 $REV $CHR

#2 discard string [] only

```
string $2
```

$2 discards the top string. It is used to discard the unwanted product of an operation such as $- or VAL.

examples

```
"hello" "there" $2          produces   "hello"

"hello there" 6 $- $2       produces   "there"
```

related words $+ $- $12 $REV $CHR

% introduce comment

% causes the rest of the line to be ignored, allowing a comment to be included. Any normal printing characters can be included in the comment, but control codes, including carriage return, cannot.

Example 3 VOICES string % for chords

& indicate hexadecimal number

&<hex digits> -> number

& goes before a hexadecimal number. The digits 0-9 and A-F are allowed in the number. There must be no spaces between digits or between & and the first digit.

examples

```
&FF     is equivalent to      255
&8000   is equivalent to   -32768

... &FF AND ...     % leave lower byte only
```

related words &NOUT

' accent next note or hit

' applies an accent to the next note or hit. It operates by temporarily adding the value set by 'L to the normal dynamic level of the next note or hit.

The interpretation of ' in sound depends on the voice type, and possibly the instrument as well.

' stops any change of dynamic level (due to +L or -L) at its current value.

example

```
'XXXX 'XXXX 'XXXX 'XXXX    % accent every 4th beat
```

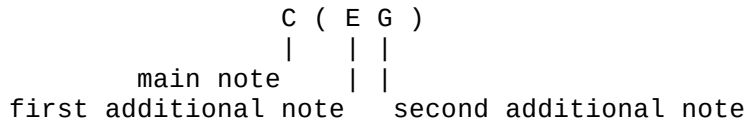
related words 'L = L

further information

' also works on rests and holds, but this has no effect on the sound when using the default music action.

(start additional chord notes
(see further information)

Chords are written using round brackets. A chord consists of a first ('main') note, followed by the other notes enclosed in round brackets. In other words, the brackets contain those notes that are to play at the same time (that is, on other voices) as the previous note.



The pitch of the notes in a chord sequence use the upper-case-up, lower-case-down rule as do notes in a simple tune, but the pitch does not simply run through all notes in the order they appear - the main notes are not affected by the additional notes. The main notes act as a simple line with the pitch moving in the normal way from main note to main note, so:

in the sequence:	c(EG)	D	(FA)	c(EG)	D	(FA)
the main note plays	c	D		c	D	

Each bracketed group of additional notes has its own line starting on the previous main note - the first additional note follows on from the main note, the second follows on from the first, and so on. At the end of the bracketed group, the last additional note does not affect the next main note.

In the example above, the additional notes rise in pitch from the main note, but they could equally well be written down, or up and down, from the main note. If the sounds of the voices are identical, that is, they have the same instrument, volume etc., it makes no differenee which voice a given note is played on. However, it's clearest to choose a single direction, either bottom up or top down, and stick to it throughout the chord sequence.

Rests and holds can be included in the main line or in the brackets to stop the sound of notes on individual voices. You do this to play a chord which has fewer notes than the one before it, for example:

```
C(EGB) F(AC^) F(AC)
```

A full chord of rests (for example ^(^^)) stops all voices, playing a rest for the whole part. A more convenient way of doing this is with the chord rest symbol, ^; .

If no symbol appears for a particular voice, it continues as if a

hold had been written. This means that a hold on the main voice will hold the complete chord, so you never need to write a full chord of holds. For example, /(//) is equivalent to / .

You can play broken chords (chords with a strumming effect produced by a delay before each additional note in the chord) by adding a length setting after the open bracket. Each subsequent note will then have a corresponding delay before it. This length only applies inside the brackets - it does not affect the length setting of the main notes. Also, the extra length of the bracketed notes does not affect the length of the main note, so it does not make the whole chord longer.

The effect of a length setting inside brackets is as follows:

```
                Normal chords           Broken chords

                48, C(EGB)F(ACE) ^;     48, C(8,EGB)F(8,ACE) ^;

   Voice 4      B-----E-----^              B-----E--^
   Voice 3      G-----C-----^             G-----C---^
   Voice 2      E-----A-----^            E-----A----^
   Voice 1      C-----F-----^           C-----F-----^
```

To make broken chords stop completely before the next strum, you can 'dampen' all voices by putting a zero-length chord rest between them, for example

```
   48,C(8,EGB) 0,^; 48,F(8,ACE) ^;
```

Hits (Xs) can be included in chords to restrike previous notes or to play percussion instruments. Even if you are using hit symbols defined to play on particular voices with ';', you can still use brackets to play two or more hits together.

Remember that before you can play chords, you must put an instrument on each of the voices you want to be heard.

examples

```
   simple sequence              c(EG)D(FA)c(EG)d(FA)
   isolated chord               C(EG-B) ^;
   elaborate sequence           C(EG) /(/a) g(BD) / f(A^)
   broken chords                C(8,EGB)F(8,ACE)
   moving main voice            C(F)bag f(D)GAB C(F)/// ^;

   percussion with X            X/ /(X)/ X/ X(X)

   percussion with user symbols "x" [1;X ]  "y" [2;X]
                                ...
```

x///y/ /x x///y//x(y)

related words); ^; , VOICE VOICES

further information

When scoring broken chords, the total delay inside the brackets can even be longer than the main note, making the strummings overlap. You can also set a negative length so that the strum goes before the main note, leading up to the beat.

	Overlapping	Pre-delay
	48,C(24,EGB)F(24,ACE)	C(-8,EGB)F(-8,ACE)^;

```
Voice 4           B-----E-----       B-----E-------^
Voice 3        G-----C-----           G-----C------^
Voice 2     E-----A-----              E-----A-----^
Voice 1  C-----F-----                 C-----F----^
```

When using overlapping and pre-delay chords, keep in mind that you are actually scoring notes to play in the future and the past, and watch out for them reaching past other events in the present. For example, if you were to add a chord rest (^;) to the end of the overlapping example above, it would silence the F, A and C, and the B from the previous chord. The final E would then play as normal.

SCORE does a '1;' so unless you use ';' yourself', normal notes will always play on voice 1, and additional chord notes will start on voice 2. You can use ';' outside the brackets to make the chord start on a higher voice.

The chord brackets achieve their function by doing two things: setting 0, so the following notes play immediately, and causing each note, hit, rest or hold inside brackets to add one to the ; setting, so they play on successive voices, starting from the main voice plus one. The , and ; settings are local, like the note pitch, so they are restored by). Inside brackets, the length of a note takes place before it starts sounding, so that by making the length greater than 0 , the notes are delayed to give broken chord effects.

The dynamic level normally applies equally to all notes in a chord, but settings can be included in the brackets to affect individual notes.

You cannot put more chord brackets or a key signature inside chord brackets.

(...) use the number stack as temporary storage for the chord-local music environment, so, strictly, their input/output descriptions are:

```
( -> octnote length tranvoice
octnote length tranvoice )
```

The temporary values are as follows:

name	low byte	high byte
tranvoice	voice (;)	transposition (@)
length (,)	low	high
octnote	effective last note	octave (:)

'(' and ')' can be classified as music events (like note, rest, hit and hold), since in addition to their modifying effects on the events contained between them, they call the music action list, from where they look like holds. '(' issues a duration to retract the duration of the last note (so inside the brackets, time starts from the main note's position), and ')' reinstates the main notes duration, correcting for the total duration of music events used inside the brackets. They pass the following stack frame:

description	value	default destination
pitch voice	OFF	VOICE
pitch	undefined	PITCH
level voice	OFF	VOICE
level	undefined	VEL
gate voice	OFF	VOICE
gate	undefined	GATE
duration	calculated duration	DURATION

On return from the action list, '(' and ')' both execute EVERY VOICE to return the voice selection to a defined state.

)

end additional chord notes

Chords are written using round brackets. See (for more information.

*

indicate operating system command command

```
*<line>
```

* indicates an operating system command. The following characters, up to the end of the line, are executed as an operating system command.

To switch to another language, leaving AMPLE, you use the normal language entry * command, for example:

```
*BASIC
```

OS commands that corrupt language memory are forbidden. These include those that have warnings in their own documentation about corrupting programs, for example *COMPACT, *BACKUP, *FORM80 (Acorn DFS), and *FX20 (OS). If you need to use, for example, a disc command that corrupts memory, you must first switch to another language such as BASIC.

```
examples        *CAT
                *FX12,4
```

related words OSCLI

further information

* is a command, so you cannot use it in word definitions. You can use the OSCLI word to issue an operating system command from a program. To issue *FX commands, you can define your own FX word - see CODE for details.

Some extension ROMs have commands or other functions that seriously corrupt language memory and interfere with AMPLE, but do not necessarily interfere with other languages such as BASIC, and have no warning of the fact in the documentation. This is often the case when the ROM uses language memory temporarily on processing a * command (either one of its own or one recognised by the OS or another ROH). Though the contents will be restored before returning, AMPLE uses its memory continuously under interrupts, so even temporary corruption can cause interference.

+ sharpen next note

+ is the sharp sign. It raises the pitch of the next note by one semitone, overriding the key signature. It only affects the next note.

```
examples        +F      % F sharp
                ++C     % C double-sharp
```

related words - = K(

further information

The total modification of the pitch can be up to plus or minus 64 semitones.

, set length

number,

',' sets the basic length for notes, hits, rests, and holds. The number is normally in the range 0 to 32767.

All notes, hits, rests and holds last for the basic length that has been set with ','. Longer events can be made without changing the basic length, by extending with the hold symbol, '/'.

Suggested basic length settings for common note values are as follows:

name	alternative name	length
hemidemisemiquaver	sixty-fourth	3
demisemiquaver	thirty-second	6
semiquaver	sixteenth	12
quaver	eighth	24
crotchet	quarter	48
minim	half	96
semibreve	whole	192
breve		384

To get the basic length for modified note values such as dotted and triplet, just multiply the basic length by the appropriate factor:

modified value	factor	example	
dotted note	3/2 (1+1/2)	crotchet	48 -> 72
double-dotted note	7/4 (1+3/4)	minim	96 -> 168
triplet note	2/3	quaver	24 -> 16

You can set short basic lengths for the grace notes in ornaments such as trills and mordents. A basic length of zero is sometimes useful, for example, on a rest which ends a note without occupying any time itself.

The basic length can be set inside chord brackets to spread the notes. See '(' for more information.

examples

```
48,         % crotchet
32,         % dotted quaver
16,         % quaver triplet
```

```
            48,C/// fG 24,A//B 48,C//B C/// ^
```

related words / \ BAR | ( )

further information

What basic length you choose to represent a particular note value such as a crotchet is entirely up to you, since the tempo can be adjusted over a wide range to give the correct playing speed. Large values are cumbersome and small values cannot be divided so far for short notes. The suggested set of values based on a crotchet of 48 allows the full set of normal and triplet values to be achieved through division and multiplication by 2 and 3. If an application needed divisions into 3,4 and 5 units for example, a crotchet of 60 would be a better choice.

The full range of the basic length is -32768 to 32767. The ability to program negative-length events is an advanced feature which can be used for articulation effects, pre-strummed chords, backwards-playing notes, overlayed note sequences, random access to the time domain, and more. The basic length translates directly into DURATION when a music event is played, so this is really a feature of DURATION. See DURATION for more information.

-

flatten next note or indicate negative number

The '-' symbol has two functions: as a minus sign for numbers, and a flat sign for notes.

Immediately before a decimal digit, it acts as a minus sign and the number is accepted as a negative decimal number.

Otherwise, it acts as a flat sign, lowering the pitch of the next note by one semitone, overriding the key signature. It only affects the next note.

examples
```
-200        % minus 200
-B          % B flat
--a         % A double-flat
```

related words + = K(

further information

The total modification of the pitch can be up to plus or minus 64 semitones.

/ hold music event

'/' is the hold symbol. It holds the last music event on each voice for one 'beat' - the length set with ','. It is used singly or repeated to extend notes, hits, and rests.

The hold is the most basic AMPLE music symbol since it simply marks a unit of musical time in which notes, hits, and rests all continue with no change. It is often used where a rest would appear in more specialised music notations, for example, between percussion hits, and in long sections where a part doesn’t play.

'/' affects chords exactly like normal notes, so a single '/’ in the main voice holds all the voices. There is no point in writing a full chord of holds, since the holds inside the brackets, and the brackets themselves (for example (///)) are redundant. You do use '/' in brackets to hold voices when a higher voice is not being held, for example C(/G)

examples

```
C/// fGAB C//A G///  % tune with simple rhythm
48,f | /             % two crotchets tied across bar line
| X/// /X/X |        % 'rests' with percussion instrument
0,^ 192,/|/|/|/|     % four bars 'rest'
C(GE) / C(AF) C(BG)  % held chord
C(EA) /(/G) a(DG)    % passing note in chord sequence
```

related words , \

further information

The length of the hold is added to the bar's total of note lengths for checking by the next bar line.

0,/ has no effect on the music. Since '(' sets '0,' a hold inside chord brackets has no effect on musical time, but may still be required to 'pass over' its voice so a note or rest can be put on a higher voice.

There is a separate word /// which does exactly the same as three separate hold symbols, and is provided only to represent groups of holds more compactly. A /// instruction uses the same amount of memory as a / instruction.

Each note letter word calls the player's current music action

list, passing the following stack frame:

description	value	default destination
pitch voice	OFF	VOICE
pitch	undefined	PITCH
level voice	OFF	VOICE
level	undefined	VEL
gate voice	OFF	VOICE
gate	undefined	GATE
duration	',' setting	DURATION

On return from the action list, it executes EVERY VOICE to return the voice selection to a defined state.

If ~ is applied to / , it is recognised and therefore cleared, but has no net effect on the interpretation.

: set octave

number :

':' sets the octave for the next note. 0: means the octave starting at middle C, positive numbers give higher octaves, and negative numbers give lower octaves.

You use ':' to set the octave at the start of each line or phrase of music. Though it can be used in the middle of a line or phrase, the '!' word is more suited for this.

examples

```
0: % centre of range, around middle C
1: % treble (G) clef, C is the third space on the stave
-1: % bass (F) clef, C is the second space on the stave

0:C/// fGAB C//A G///      % play tune at middle C
```

related words !

further information

':' actually sets the effective last note pitch to be C in the numbered octave, so for example 0:C sets the pitch as if middle C had just played, with an upper case C. Since you can follow the ':' by a lower-case letter, a particular octave number gives access to two octaves of pitches:

```
0:c 0:d 0:e 0:f 0:g 0:a 0:b 0:C 0:D 0:E 0:F 0:G 0:A 0:B
<-----lower octave--------> <------upper octave------->
```

;

set music voice

```
voicenumber ;
```

';' sets the voice on which notes, hits and rests will play. For chords, you normally use the chord brackets to automatically place the notes within on successive voices, but in other cases it is more convenient to specify the voice directly with ';', for example, for

* a line that plays on one voice while other voices are held
* complex sequences of overlapping notes
* playing on all voices simultaneously
* a user-defined percussion word that hits a particular voice

Since the length of a note, rest or hit affects all voices on the player (like a hold), the voices always stay in step. A note, hit or rest played on one voice sounds until another one on the same voice, even though you may have played on other voices in the meantime. A hold has the same effect whatever voice it is on.

SCORE does a '1;' so unless you use ';' yourself, normal notes will always play on voice 1, and additional chord notes will start on voice 2.

The upper/lower case pitch movement is common for all voices on a player, so the pitch moves from note to note as normal, whatever the voice.

Remember that you must put an instrument on each of the voices you want to play.

examples

```
1;                          % use voice 1

16,1;cDEF G 2;fGA B 3;aBCD  % chord build-up effect

1;C 2;G 1;D 2;A 1;E 2;B     % overlapping notes

"x" [1;X ]  "y" [2;X]       % user-defined hit symbols...
...
x///y//x x///y//X(y)        % ...used in percussion score
```

related words (VOICE VOICES

= naturalise next note

= is the natural sign. It cancels the effect of the key signature on the next note, so it plays at its unmodified pitch. It affects the next note only.

example
```
=b    % B natural
=F    % F naturaI
```

related words + - K( )K

@ set transposition in semitones

transnumber @

'@' sets the transposition to be applied to notes. The number before it is the transposition in semitones, which can be positive or negative.

'@' has a variety of uses, including:

* scoring music for transposing instruments
* playing a fixed note pattern ('riff') at different pitches
* moving a part to play at an octave above or below.

The following table shows the @ value (number of semitones) for transposing from C:

Transpose C to lower pitch:

Pitch name	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
AMPLE name	c	+c	d	+d	e	f	+f	g	+g	a	+a	b	C
@ value	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	0

Transpose C to higher pitch:

Pitch name	C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C
AMPLE name	C	+C	D	+D	E	F	+F	G	+G	A	+A	B	!C
@ value	0	1	2	3	4	5	6	7	8	9	10	11	12

SCORE cancels the transposition, that is, it carries out 0@ .

The key signature works on the note letters rather than the note pitches, so you don't need to worry about the effect of transposition on it.

examples

```
SCORE 12@ ...            % play music an octave up
```

```
0@ riff 7@ riff 5@ riff       % play at root, fifth, fourth

SCORE -2@                     % play music for a Bb clarinet
```

further information

The @ word can be used as a means to specify note pitch by number rather than letter, for example:

```
   the sequence    0@ 0:C  1@ 0:C  2@ 0:C   3@ 0:C ...
   plays              0:C      +C       D       +D ...

   24, 12 FOR( COUNT @ 0:C )FOR       % rising chromatic scale

   12, REP( 10 RANDL @ 0:C )FOR       % random 'robot speech'
```

The instructions to play a transposed note can be conveniently defined as a word:

```
   "note" [ @ 0:C ]
   ...
   0 note  1 note  2 note  3 note

   24, 12 FOR( COUNT note )FOR        % rising chromatic scale
```

This is particularly useful for programs that generate music from calculation or stored data, rather than fixed score.

[start word definition command

```
   namestring [...]
```

The [and] (square bracket) symbols are used to define user words.

```
                              "wordname" [ ... ]
                                  |      |  |  |
    name of word in double quotes       |  |  |
   open square bracket starts definition   |  |
                                    contents   |
          close square bracket ends definition
```

The name can be up to and including 15 characters long. Any characters may be used, but to avoid confusion with system words and sequences of them, upper-case letters and spaces should normally be avoided.

If a user word of the same name already exists, it is replaced by the new definition.

The word definition can use any existing user or system word, other than system commands (words with the 'command' status). The definition can stretch over a number of lines.

You will normally enter a new word definition using an AMPLE editor, but you can enter it directly at the % prompt. Between [and], the normal prompt is replaced by [% .

examples

```
"tune" [ 0:48,C///fGAB C//aC//a C/// ]
"cube" [ #11 #11 #* #* ] % number cube -> number^3
"x" [ 1;X ]              % user-defined music symbol
```

related words]

further information

When an existing word is redefined, players 1-10 and all sounds are stopped, and player 0's music action chain is reset.

As with other commands, the name string is in fact supplied as the second item down on the string stack (see '""').

Choosing a name that is a valid sequence of music symbols will cause confusion, for example "cdef" or ”c1". This rarely happens in practice unless short names are used, but you should still remember not to use names that use just the letters 'a' to 'g' with or without a number at the end.

To define a recursive word, you first have to create a dummy (non-recursive) definition, so that the word already exists when you try to define the real recursive one, for example:

```
"fact" []                  % dummy definition
"fact" [                   % real definition
#11 1 #= IF( #2 1 )ELSE(
#11 1 #- fact #*  )IF      % refers to dummy, which is
]                          % then replaced by real one
```

To delete a recursive word, you must reverse the process since DELETE will only delete words that are unused (not referenced by any word):

```
"fact" []
"fact"DELETE
```

\ move back

\ (back hold) is the converse of / – it moves back in music time by the basic length setting. This back-spacing effectively undoes the net time effect of previous holds, notes, rests and ties, but doesn't affect when they play, only affecting when the following notes play. It is useful when you find it more convenient to score notes in an order other than that in which they play, for example, to overlay two separately scored note sequences.

The back hold symbol lets you reach from one point in the score, say, where you have used a music symbol of your own definition, to place a note at another point in the past or future. One example of this is where a part is broken up into sections and defined as separate words. If a verse was to start with a few notes which actually played before the start of the first bar, that is in the last bar of the previous section (a pick-up), you could score it at the start of the verse where it belongs by first back-spacing with the back hold symbol.

When moving backwards and forwards in time with \ and / , watch out for spacing past other sound events like instrument changes.

In all other respects, including the effect inside chords, \ is exactly like / . In fact, 48,\ is equivalent to -48,/ , and -48,\ is equivalent to 48,/ .

examples

```
    /\   or   \/                % do nothing

    12,\\\dEF G/// /fed c///    % three-note pickup

    24,x/// x///           x///  % percussion score ...
    24,x/// x//20,/ 4,x 24,x///  % with flam (double hit) ...
    24,x/// x///   4,\x 24,x///  % then written with \

    % interleave notes of two riffs, half a beat apart
    riff1 48,\\\\ \\\\          % back to start
    24,/ riff2 24,              % forward half beat, then back
```

related words / , ( )

further information

There is no limit to the amount of time by which you can back-space with \ , but there is a limit to the number of sound events you back-space past. This is the same as for DURATION – see DURATION for details.

] end word definition [] only

The [and] (square bracket) symbols are used to define user words.

See] for more infovmation.

~ slur next note

~ causes the following note to be slurred, that is, played as a continuation of the previous note without re-gating. You put a slur between two notes that you want to be played in a smooth connected fashion, with just the change of pitch between them.

The exact effect effect of a slur depends on the type of voice (some cannot carry out the slur at all), and the envelopes of the instrument. On envelopes with a non-zero sustain level, it sounds as you would expect, but on those that decay to zero, the second note is quieter and may be lost entirely. Note that 'slurs' on piano-type sounds are in fact gate period effects and are not produced with the slur symbol.

Because the slur causes the note to play without being gated, other effects of the gate, such as a 'Len' gap, are also disabled, so the only effect is a change of pitch.

Individual notes of a chord may be slurred.

examples

melody	Dc b~ggd~GG B~gBD~cb
chords	A(CE) /(/~F) A(DF) ~C(~E~G) ^;

related words A-G a-g

further information

~ takes effect by disabling the level and gate of the next music event, which it does by setting the level voice and gate voice to OFF, leaving only the pitch and duration to be executed. This works equally on note, hit, rest, and hold, though in the case of a hit, rest or hold, the result is interpreted by the default music action simply as a hold.

See 'A to G' for a description of the effect of ~ on the music action variables.

0 to 9 decimal digits

<decimal digits> -> number

A groups of decimal digits is accepted as a number whose value is then left on the stack. A preceding '-' sign makes the number negative.

The range of number values is -32768 to 32767.

examples 0 42 -273

related words & –

^ play rest

^ is the symbol for a rest. It plays a rest lasting for the basic length, on the current music voice. It finishes the previous note on that voice, causing the sound to stop or die away depending on the instrument in use.

Rests can be extended in length by the hold symbol. This is preferable to repeating the rest, for example,^/// is better than ^^^^. Hold symbols should also be used rather than rests to mark passages where a part doesn't play. In percussion scores, normal non-hit beats ('rests' in other notations) are marked by hold symbols: the rest has the effect of cutting short the sound of the previous hit, just as it ends the sound of a note.

Since rests work on individual voices, to silence all notes of a chord, you should use ^;

Each part of a piece usually ends with rests to finish the last note(s). Section ends can also include rests so that the notes always finish whatever is notated at the beginning of the next section. In this case, they must be zero-length so they take up no musical time.

examples

```
C^C^C^C^            % isolated notes
192,D | ^           % end of piece
x///x^/x x///x^x^   % decaying and cut-short beats
CEGC 0,^            % zero-length rest
```

related words ^;

further information

The length of the rest is added to the bar's total of note lengths for checking by the next bar line.

^ calls the player's current music action list, passing the following stack frame:

description	value	default destination
pitch voice	OFF	VOICE
pitch	undefined	PITCH
level voice	OFF	VOICE
level	undefined	VEL
gate voice	event voice	VOICE
gate	OFF	GATE
duration	',' setting	DURATION

On return from the action list, it executes EVERY VOICE to return the voice selection to a defined state.

If ~ is applied to ^ , it reduces it to a hold (with the default music action) – see ~ for details.

^; play chord rest

^; plays a chord rest (a rest on all voices of the part), lasting for the length setting. It is equivalent to ^(^...^) with a rest symbol for each of the voices, but more convenient for use with chords.

example C(EG-BC)^;C(EG-BC)^;

related words ^ VOICES ;

further information

^; is equivalent to EVERY; ^ , but it does not change the current voice setting. It has the effect of ^ on each voice in the range of voices set with VOICES.

| mark end of bar

| represents a bar line. Bar lines are used only to check the length of bars (detecting extra or missing items) and have no effect on the music.

Their use is entirely optional. For full scores translated from written music with bar lines, and music with many basic length changes, they are usually worth using. For simple tunes and music composed directly into AMPLE, they are usually best left out.

The first bar starts with the SCORE, so there should be no bar line before it. Bar lines are placed at regular intervals throughout the part, the last one going at the end but before any extra rest you have added to finish the final note.

BAR sets the desired bar length, and if a bar line finds that the total of lengths since the last bar line or SCORE is different from this, then it gives the 'Bad bar' error.

If the bar length is set to zero, lengths are still totalled but checking is disabled. If you are trying out extracts at the keyboard by cursor-copying incomplete bars, you should set 0 BAR to avoid unwanted bar errors.

SCORE sets the bar length to zero, so you must use BAR if bar checking is required.

examples

```
SCORE 48, 4 BAR  C E F c |      % ok
SCORE 48, 4 BAR  C E D |        % gives 'Bad bar'
SCORE     0 BAR  C E D |        % not faulted
```

related words BAR

A to G play note with ascending pitch

The letters 'A' to 'G' play notes of their respective pitches above the previous pitch.

AMPLE music notation uses the letters A-G to represent note pitches of the same names (the white keys on a piano keyboard). In upper-case (A-G), the letter plays that pitch above the last pitch, and in lower-case (a-g), it plays the pitch below. The exception is that a repeated letter (with the same case) always plays the same pitch.

This effect allow a phrase of music to be written using note letters without octave indications: each note's pitch is up or down from the previous one, depending on its case. The octave pitch of the whole phrase is set with a single ':' at the start. For intervals greater than an octave, '!' causes an extra octave jump on the following note.

The note takes as its length the basic length (set with ',') though you can extend it with the hold symbol (/).

The '+'(sharp) and '–' (flat) symbols modify the pitch of the following note by one semitone. A key signature (K(...)K containing a list of sharpened or flattened notes) modifies all

uses of particular note letters, except those that have a '+', '-' or '=' (natural) symbol on them. @ transposes all note pitches by a specified number of semitones.

Notes normally play on the current music voice (set by ;), but inside chord brackets ((...)), they play on successive voices starting on the current music voice. Chord brackets also set the length to zero temporarily, so the notes start at the same time.

examples

```
CDEFGAB^              % rising scale
Cbagfedc^             % falling scale
CCCC^                 % repeated note
cCcCcCcC^             % alternating octaves
CCDbCD EEFedc DcbC^   % pitch sequence of phrase of
                      % 'God Save the Queen'
```

related words ! : + – = K(@ ~

further information

The length of the note is added to the bar's total of note lengths for checking by the next bar line.

Also, see '('.

Each note letter word calls the player's current music action list, passing the following stack frame:

description	value (if normal)	value (if after ~)	default destination
pitch voice	event voice	event voice	VOICE
pitch	calculated pitch	calculated pitch	PITCH
level voice	event voice	OFF	VOICE
level	calculated level	calculated level	VEL
gate voice	event voice	OFF	VOICE
gate	ON	ON	GATE
duration	',' setting	',' setting	DURATION

On return from the action list, it executes EVERY VOICE to return the voice selection to a defined state.

a to g play note with descending pitch

The letters 'a' to 'g' play notes of their respective pitches below the previous pitch.

See 'A to G' for more information.

ACT execute music action

```
see text
```

ACT calls the next in the player's chain of music actions. It is may be used inside ACT(...)ACT in the definition of a music action to modify the interpretation of music events, or outside ACT(...)ACT to call the music action chain to generate music events directly.

ACT accepts a seven-number description of the music event in a stack fr ame:

```
dur gate gatev level levelv pitch pitch FRAME ACT ->
dur gate gatev level levelv pitch pitch FRAME
```

When the music action chain is empty (that is, no actions are in use) ACT calls the following 'terminating' routine to feed the seven numbers (the **action variables**) to the standard 'music playing' sound words that are provided for all voice types:

```
7 FCOPY
% duration gate gatevoice level levelvoice pitch pitchvoice -> 0
 VOICE PITCH
 VOICE VEL
 VOICE GATE
 DURATION
```

Separate voice settings for each sound event allow each sound event to be sent to the destination voice or disabled (with OFF VOICE) as required by that music event.

ACT(...)ACT supplies the action variables for any ACT included in the sequence, for example:

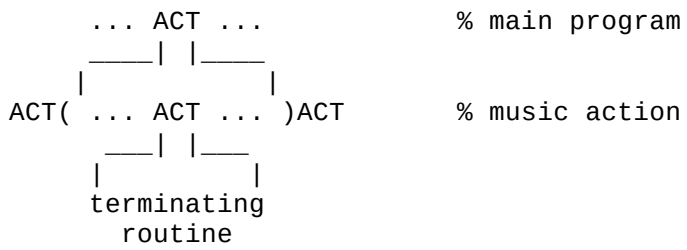
```
ACT( ACT )ACT
```

In contrast, where ACT is used in the main program, the program must supply the variables for example:

```
48  ON 1  64 1  0  1  FRAME  ACT
#2  #2 #2 #2 #2 #2 #2
```

This will play a note with pitch 0, level 64, and duration 48.

When the music action chain has one or more items on it, each ACT instruction calls the action that is next in the chain from its own position. Successive actions use ACT to pass control along the chain, until in the last action, ACT calls the terminating routine as described above. Each ACT enters the next action at the first instruction after ACT(, and is returned to when the action reaches)ACT. For example, with one action in the chain, control follows this course:



Some actions may deliberately not include an ACT, thereby ending the event's interpretation as an alternative to the standard terminating routine. Others may call ACT more than once so that a single input music event is interpreted as two or more output music events – to create an echo, for example.

Actions may read and write the action variables in their stack frame using FVAR as normal:

	function	sound word
1 FVAR	pitch voice	VOICE
2 FVAR	pitch	PITCH
3 FVAR	level voice	VOICE
4 FVAR	level	VEL
5 FVAR	gate voice	VOICE
6 FVAR	gate	GATE
7 FVAR	duration	DURATION

Many actions read the action variables to control additional output (possibly through ACT or direct sound words), and/or write them to alter the results of the terminating routine and down-chain (later) actions. Actions normally preserve the action variables, copying them first with FCOPY if they need to pass modified values to ACT. Others may change the original values, either to send results back to the previous actions or main program, or simply to maximise speed by not copying, accepting that this might affect the operation of any up-chain (earlier) action that expected values to be preserved.

ACT is the originator of all music events, whether from Nucleus music event words, user programs, or additional module functions including real-time music event generators such as a music keyboard and other input devices. Real-time music events differ from other ('advance') music events only in their use of the duration variable (7 FVAR), which sends the duration of the **previous** event (calculated from QTIME), in arrears. Simple action definitions are entirely compatible with real-time events, but because real-time events always operate 'behind time', definitions that use DURATION or other than a single ACT require re-arrangement.

examples

```
"randn" [ % play random-pitch notes using any effects available
READY 1 VOICES instrument
...                          % add any music action effects here
REP(
 48                          % duration
 ON 1                        % gate and voice
 0 OFF                       % no level (sent to 0 VOICE)
 12 RANDL 1                  % pitch and voice
 FRAME ACT                   % execute
 #2  #2 #2  #2 #2  #2 #2
)REP ]

"randn" [ % alternative
READY 1 VOICES instrument
...                          % add any music action effects here
48                           % duration
ON 1                         % gate and voice
0 OFF                        % no level (sent to 0 VOICE)
0 1                          % pitch and voice
FRAME
REP(
 12 RANDL 2 FVAR #!          % set pitch
 ACT                         % execute (assuming preservation)
)REP ]

"noop" [
20 ACT( ACT )ACT ]           % basic 'no effect' action

"sink" [ 20 ACT( )ACT ]      % throws all music events away

"repeat" [ % duplicates each music event
20 ACT(
 ACT                         % do it
 ACT                         % do it (including duration) again
)ACT ]
```

```
"mon" [ % action monitor – print variables on each event
30 ACT(
  "   v pitch v level v gate duration"
  $OUT NL                  % print legend
  7 FOR( COUNT FVAR #?     % get each action variable
    $STR 4 $PAD $OUT       % and print (in field of four)
  )FOR NL
  ACT                      % continue with the event as normal
)ACT ]
SCORE mon
C                          % prints
                           &   v pitch v level v gate duration
                           %   1   0   1  64   1   -1  48
```

related words ACT(FRAME

further information

The standard set of seven action variables is not a feature of ACT itself, but merely of the Nucleus standard terminating routine and music event generators. The user can easily implement an alternative music event standard, possibly with more values for greater control, having its own music event generators and terminating routine, and/or actions to convert to and from Nucleus standard music events.

ACT(

start music action sequence [] only

positionnumber ACT(

ACT(...)ACT defines a music action. It either adds the enclosed sequence to the player's music action chain at the position specified, or, if the position is zero, it removes the action from the chain.

ACT(is used to set a sequence of instructions to be carried out on each music event (note, hit, rest, or tie), and hence to redefine the interpretation of music events.

When the music action chain is called, control passes to the first action in the chain, entering at the first instruction after ACT(. Control returns to the calling routine when)ACT is reached. The sequence may call the next in the chain through the word ACT.

Position numbers between 1 and 127 determine where in the chain the action is placed – a low number goes before a higher number. Where there are two actions with the same position number, the most-recently selected is placed first. Each action can only be included once in each player's chain – on inserting it a second

time, the first is removed. Position number 0 removes the action from the chain.

A convention applies to the choice of position number for action that need to be compatible with others for simultaneous use. Most action definitions fall in to one of three classes, each of which has a single position number assigned by the convention:

position number	class	type of processing	example
10	early	score	set transpose to pitch of note
20	middle	music	generate echoes of each event
30	late	sound	change sound depending on pitc

Score processing actions set or respond to music environment variables, and include those that redefine music event words for notational purposes. Such sequences often use music environment words to set, and MVAL? to respond. If the normal ACT is left out of the sequence, events will not play. A temporary action may turn itself off with a self-reference once it has completed.

Music processing actions respond to and issue standard music events to perform operations such as expansion, both on the same voice, for example converting single notes to sequences, and on other voices, for example playing true echoes or parallel voices. They use ACT both directly, and indirectly through music events words included in the action sequence itself.

Sound processing actions use sound words (including voice-type specific ones) to control voices directly, with or without invoking the default interpretation through ACT. Examples include articulations, automatic panning, pitch slides etc. Sequences often read the action variables (with FVAR) to determine the type of event and set the voice for sound words.

examples

```
% fixed pitch: all letters play on 0: (4-oct range using !)
"fix" [
10 ACT(            % early (for score processing)
ACT                % do event as normal
0:                 % then set octave for next note
) ACT
0: ]               % set octave for first letter after 'fix'.
...
SCORE fix CDEFGAB!CBAGFEDC^  % up-and-down scale in 0:

% alternative, with control flag
"fix" [ % flag fix
```

```
        % ON fix -> fixed pitch mode
        % OFF fix -> normal pitch mode
10
AND                % 10 if ON, 0 if OFF
ACT( ACT 0: )ACT   % all events do 0: for the next one
0: ]               % for the very first event

"macro" [ % expand each note to a sequence on that pitch
20 ACT(            % middle (for music processing)
  MVAL?            % save music environment
  2 FVAR #? @      % set transpose to pitch
  0:CCGG           % play riff (using global ',')
  MVAL!            % restore music environment
)ACT ]
SCORE macro
48,CD              % plays 48,CCGG ddAA
12,CFFggCC

"dance" [ % set random stereo pan on each event
          % (for voice types that have PAN)
30 ACT(                  % late (for sound processing)
  1 VOICE                % select voice (
  6 RANDL 3 #- PAN       % set random pan
  ACT                    % execute event as normal
)ACT ]
SCORE dance 12,CCCCCCCC

"dance" [ % improved version which selects voice and
          % only acts on ON GATE's, i.e., notes/hits
30 ACT(                  % late ( for sound processing)
  5 FVAR #?              % get gate voice: OFF, 1, 2, 3 etc
  6 FVAR #?              % get gate value: OFF or ON
  AND                    % voice number if ON GATE,
  VOICE                  % OFF if OFF GATE/no gate
  6 RANDL 3 #- PAN       % set pan (no effect if OFF VOICE)
ACT ) ACT                % always execute event as normal
]
SCORE dance
12, C^/CC//CC^/CC///^

% phrasing – slurs every note in a contiguous note sequence
"prevgate" [GVAR]
"phrase" [
20 ACT(
  5 FVAR #? IF(          % if gate present (gatevoice <> OFF)
    6 FVAR #? IF(        % if gate ON
      prevgate #?        % if previous gate was ON
      IF( OFF 5 FVAR #!  % set this gate OFF
    )IF
  )IF
```

```
    6 FVAR #? prevgate #!  % record for next
    )IF
  ACT )ACT
  OFF prevgate #! ]        % initialise
  SCORE phrase 24,         % use instru with peak AND sustain
  aBCDEdc^Dcb^Cba^         % three separate phrases

  % phrase mark instruction for use with 'phrase' above
  "ph" [ OFF prevgate #! ] % force phrase start (without rest)
  SCORE phrase 24,         % phrasing as before, but
  aBCDEdc/ ph Dcb/ ph Cba/ ^  % without rests between phrases
```

related words ACT FVAR VOICE! FRAME SIMPLEACT

further information

An instance of ACT(...)ACT is known as an **action structure**, and the instructions it encloses as an **action sequence**.

An action structure may be used inside an action sequence itself, for example:

```
  % tran instruction: transpose to pitch of next letter
  "tranact" []       % null defn for self-reference
  "tranact" [ % flag tranact
  10 AND ACT(        % early action, controlled by flag
    2 FVAR #? @      % set transposition to pitch
    OFF tranact      % remove music action
  )ACT ]
  "tran" [
  0@                 % cancel transposition
  ON tranact ]       % turn on music action
  ...
  tran 0:-b ...      % transpose C -> -b
  tran 1:C           % transpose C -> ! C (up octave)
  tran 0:C           % cancel transposition
```

)ACT end music action sequence [] only

ACT(...)ACT defines a music action. See ACT(.

ALIGN ensure text cursor is at start or line

ALIGN makes sure that the text cursor is at the beginning of a line, that is, in column zero • If the cursor is not in column zero already, ALIGN moves it to the start of the next line by printing a carriage return/line feed.

example ALIGN "Enter pos:" $OUT % put prompt at line start

related words NL

AMPLE restart system command

The command AMPLE restarts the system, clearing the program to make ready for entr y of a new program. The rest of the input line is ignored.

It also removes P and T modules, QUITs the editor and checks that the Nucleus ROM image is complete, issuing the '! Bad ROM' error if not. F modules are retained.

further information

The AMPLE command also selects and initialises the time-server with the alphabetically latest name.

AND AND bits of numbers

number1 number2 AND -> ANDnumber (number1 AND number2)

AND performs the logical AND operation on the bit patterns of the two numbers. Each bit in the result is only 1 if both corresponding bits in the input numbers are 1.

AND is used both as a bit-wise operator for manipulating bit patterns, and as a logical operator for flags:

bit1	bit2	bit3	flagl	flag2	flag3
0	0	0	OFF	OFF	OFF
0	1	0	OFF	ON	OFF
1	0	0	ON	OFF	OFF
1	1	1	ON	ON	ON

example

```
&1234 &FF AND    produces   &34
in binary:
          0001001000110100 AND
          0000000011111111
produces  0000000000110100

#11 0 #> #12 5 #< AND      % number -> ON if 1<=n<=4, else OFF
```

related words OR XOR NOT

ARRAY access array element

elementnumber baseaddressnumber ARRAY -> addressnumber

ARRAY is used in the definition of arrays. It is usually used after DIM, which reserves store from program memory, in a word definition which serves to give a name to the array.

```
              "totals" [ 10 DIM ARRAY ]
               |         |
      array name         |
    maximum element number (number of elements is 11)
```

The array elements are numbered 0 to the number specified, though you will often leave 0 unused. Array elements are stored to (set) and fetched from (read) by #! and #?. The array element number goes immediately before the array name.

As an alternative to standard arrays, which take a single argument, it is possible to create internally indexed arrays that take no arguments and are used like variables, and multiply indexed arrays which take two or more arguments. In these cases, the single element number required by ARRAY is provided by additional instructions added at the start of the standard array definition.

examples

```
"values" [ 20 DIM ARRAY ]
... 0  1 values #!              % store 0 in element 1
... 3 values #? NOUT            % print element 3

                                % 10 x 10 array
"matrix" [ % col row matrix -> address
1 #- 10 #* #+                   % element = (row-1)*10 + col
100 DIM ARRAY ]                 % col row matrix -> address
....
                                % print matrix
10 FOR( COUNT                   % column loop
 10 FOR(                        % row loop
  #11 COUNT matrix #? NOUT SP   % use column number on stack
 )FOR #2 NL                     % discard column number
)FOR

"plvar" [ % plvar -> address    % player-local variable
PNUM                            % index is player number
10 DIM ARRAY ]                  % normal array definition
....
```

```
    0 plvar 1!                      % used like normal variable
    ....                            % but is independent for
    plvar #? NOUT                   % each player

    "crazy" [                       % random shuffler!
    8 RANDL                         % random index value
    8 DIM ARRAY ]                   % normal array definition
```

related words DIM #! #? #+! #B! #B?

further information

ARRAY performs the following:

1 Converts the element number into the location address
2 Checks that the address is in range

ARRAY is sometimes worth using where only a DIM is required, merely for its range checking function. It can later be removed for speed when the program is fully tested.

Any arrays created by a program remain in existence until discarded by those commands that stop any program that is <ecuting, that is, COMPACT, SAVE, LOAD, NEW etc.

ASC convert character to number [] only

```
   string ASC -> asciinumber
```

ASC converts a character into its ASCII code, a number. If the string is longer than one character, the trailing characters are ignored. If the string is a null string, ASC returns -1.

examples

```
   "A" ASC   produces   65

   ...                   % string on stack
   1 $- ASC #11 $CHR $+  % find ASC but retain
   ...                   % string and number on stack
```

related words $CHR

#B! store low byte of number at address

```
datanumber addressnumber #B!
```

#B! stores a single byte (the low byte of the number) at the address. #B! is often used with single-byte arrays.

It will operate on any address and should therefore be used with care to avoid corrupting memory. exaaples

examples

```
1 &8E #B!          % store 0 at location &8E

0 buffer 3 #+ #B!  % store 0 at buffer+3
```

related words #B? #! #? DIM

further information

Two-byte values are stored with the low byte at the address and the high byte at the address plus one, for example:

```
1 xvar #B!          % store 1 in low byte of variable xvar
0 xvar 1 #+ #B!     % store 0 in high byte of xvar
```

#B12 swap high and low bytes of number

```
number1 #B12 -> number2
```

#B12 swaps the high and low bytes of the number. It is used for byte-processing operations.

It can be used as a fast '256 #*' (placing the top byte) and an unsigned '256 #/ #2' (extracting the top byte – not possible with the signed #/). These operations are useful with CODE, that uses the high and low bytes of numbers independently.

examples

```
&1234 #B12   goes to   &3412

... CODE    % YX PA   two number returned by code
#2                    % YX
#11 #B12 &FF AND      % YX y
#12 &FF AND % Y X     Y and X as separate numbers
```

#B? fetch byte from address

```
addressnumber #B? -> datanumber
```

#B? fetches the byte from the address. The number has the fetched byte as the low byte, and the high byte is zero. #B? is often used with single-byte arrays.

examples

```
&8E #B?            % fetch and print byte as &8E

value 1 #+ #B?     % fetch high byte of word at 'value'
```

related words #B! #! #? DIM

further information

Two-byte values are stored with the low byte at the address and the high byte at the address plus one. #B? can be used to extract the separate byte from a variable:

```
var #B?             % fetch low byte
var 1 1+ #B?        % fetch high byte
```

BAR set bar length

```
beatsnumber BAR
```

BAR sets the bar length in beats. The beat is usually specified beforehand with a length setting, so the two act as a time signature: the length setting has the function of the bottom number, and the BAR setting is the top number. Only the total bar length is important – the number and length of beat have no additional effect.

After a BAR setting, if any bar in the music does not add-up to the same total length, the 'Bad bar' error is issued at the bar line.

With the bar length set to 0, the checking action of bar lines is disabled. SCORE does a 0 BAR, so you must make a BAR setting if you want bar length to be checked.

The BAR setting has no effect on the sound of the music.

See | for more information.

examples

```
48, 4 BAR   % 4/4 (4 crotchets per bar)
```

```
   92, 2 BAR   % 2/2 (2 minims per bar), same effect as 48, 4 BAR

   24, 5 BAR   % 5/8 (5 quavers per bar )
```

related words |

further information

BAR may be used at any point in the music, for example, for a new time signature. BAR does not interfere with length totalling – a new setting can only take effect at the next bar line.

CLEAR clear editor data command

CLEAR clears the public editor data, freeing the memory for re-use. It is used to clear the data before entering new data or saving the program. After CLEAR, SHOW shows 'no data'.

Editors that use private data storage, rather than public, replace the Nucleus CLEAR command with one of their own while selected. This clears the editor's private data, and has no effect on public data. QUIT exits the editor, reinstating the Nucleus CLEAR.

&CHR convert number to character [] only

```
   asciinumber $CHR -> string
```

$CHR converts the number to the corresponding ASCII one-character string. If the number is negative, a null string is produced. Values between 0 and 255 are allowed.

$CHR is often used in assembling strings from numbers, and in particular, for putting control codes in strings.

examples

```
   65 $CHR   produces   "A"

   13 $CHR   produces   "<cr>" (carriage return>

   -1 $CHR   produces   "" (null string)

   "instant" [ % key press -> command, e.g. A-g and a-g play notes
   "instant" $+    % return to this word when done
   #IN $CHR        % get key and convert code to string
   $+              % add on to command line
   0, ]            % make all notes immediate
   READY 1 VOICES instrument
   instant
```

CODE call machine-code routine

```
YXnumber CAnumber addressnumber CODE -> YXnumber PAnumber
```

CODE calls the machine code routine at the address given, and returns control when the routine exits by RTS. It takes two numbers to set the processor registers on entry (YX and CA), and returns two numbers (YX and PA) with the register contents on exit:

On entry		On exit	
A	low byte of CA	A	low byte of PA
C	bit 8 of CA	P	high byte of PA
Y	high byte of YX	Y	high byte of YX
X	low byte of YX	X	low byte of YX

CODE may be used for calling operating system and user routines. Where a data block is required, memory can be reserved for it with DIM – see DIM for details.

examples

```
1 15 &FFF4 CODE #2 #2 % *FX15,1 flush input buffer

"FX" [ % YXnumber CAnumber FX
&FFF4 CODE #2 #2 ]
...
1 15 FX      % flush input buffer
```

further information

User routines may be accommodated in unused operating system workspace, ROM, or memory reserved with DIM.

Zero page locations &8E and &8F are available for use by user routines.

The routine is entered with the I, D and B flags clear.

COMPACT compact unused memory command

COMPACT arranges unused memory space into one contiguous area, making it fully available for use.

In general use, the space freed by deleted words and other items can be left in isolated pieces which are too small for re-use. When this happens, an operation may fail with the 'No room' error when there is in fact enough space in total, particularly when only a small amount of user memory is free. In this event, you

may enter COMPACT and re-try the operation that gave the error.

See MEM for guidance on memory economy.

On computer models without shadow RAM, you can also get a 'No room' error when changing to a lower-numbered screen mode (mode 7 to mode 3, for example) since this incurs a reduction in the amount of language memory. If this happens, use COMPACT and try again.

related words MODE MEM

further information

COMPACT stops players 1-10 and all sounds, and resets player 0's music action chain.

Programs are saved in a compacted state, so immediately after SAVE or LOAD, the free memory is compacted. This means that you can include a MODE change in a program and be sure that you won't get a needless 'No room' when it is LOADed and RUN.

COUNT return loop count [] only

```
COUNT -> number
```

COUNT leaves the loop count of the most recent FOR(...)FOR loop containing it. The count starts at one and increases by one each time around the loop. On the last time around, it is equal to the number given to FOR(

COUNT lets the instructions inside the loop do something different on each pass, usually sequencing through a particular range of values.

The COUNT can be put anywhere inside the FOR(...)FOR loop, except enclosed in a separate word ([...]) or intervening P(...)P structure.

examples

```
"countup" [ % print numbers from 1 up to 20
20 FOR( COUNT NOUT SP )FOR NL ]

"timestab" [ % print 10x10 times table
10 FOR( COUNT   % get y count
  10 FOR(
    #11         % copy y count
    COUNT #*    % calulate x * y
    $STR 4 $PAD $OUT    % print in field of 4
```

```
    )FOR
    #2            % discard y count
    NL            % print new line for next row
  )FOR ]
```

related words FOR(INDEX

further information

COUNT counts up whereas INDEX counts down. INDEX is slightly faster in execution than COUNT.

DELETE delete word command

namestring DELETE

DELETE removes the named user word, freeing its space for re-use. It should be used with care as its action cannot be reversed.

If the user word is in use by any word, it is not deleted and the 'In use' error is given. FIND can be used to locate the uses of the word.

example "oldword" DELETE

further information

DELETE stops players 1-10 and all sounds, and resets player 0's music action chain.

To delete a self-referencing word, you must first redefine it to a non-self-referencing version, such as an empty definition: []
For example:

```
  %"fact"DELETE
! In use           % fails due to use inside itself
  %"fact" [ ]      % redefined, leaving fact unused by any word
  %"fact"DELETE    % succeeds
  %
```

If you press BREAK while DELETE is operating, the program may be left in an intermediate unexecutable state, but it will be restored on saving.

When short of free memory, you should use COMPACT to make best use of the memory freed by DELETE.

DIM reserve memory

sizenumber DIM -> addressnumber

DIM reserves a specified amount of memory for data storage by the program, and leaves the address of the first location. It is often used with ARRAY to define an array, in which case the number before DIM is the maximum array element number (see ARRAY for details) .

DIM can also be used without ARRAY, leaving the user to address the memory directly. The DIM instruction is included in a simple definition which serves to give the memory block a name. The block is (sizenumber+1)*2 bytes in size, that is, locations 'addressnumber' to 'addressnumber+sizenumber*2+1' inclusive. The definition can also carry out additional address calcultions, using input values if required. Hence the user can create custom storage words for various functions including single byte arrays, faster access store, specially-indexed arrays, string variables, operating system inter face control blocks and many others.

Note that it is the user's responsibility to check addresses against the bounds of the DIM block, if required. Checking can be included in the DIM definition, and then removed to increase speed and free memory when the progr am is complete and tested.

READY clears records of all memory reserved with DIM, so the first use of a DIM instruction after READY reserves memory afresh, not necessarily at the same address as previously. It is good practice to include READY at the start of any pr ogram that uses DIM.

examples

```
"list" [ 10 DIM ARRAY ] % array with elements 0 to 10

"chars" [ % elementnumber chars -> addressnumber
4 DIM#+ ]               % byte array with elements 0-9
...
65 0 chars #B!          % set element 0 to 65
...
0 chars #B? NOUT        % print element 0

"chars" [ % elementnumber chars -> addressnumber
1 #+ 4 DIM #+ ]    % alternative with elements numbered 1-10

"chars" [               % bounds-checking version
#11 0 #<                % less than 0?
#212 9 #> OR            % or greater than 9?
IF(
 "!! bad element in chars"$OUT
```

```
 STOP )IF               % print message and stop, or
4 DIM #+ ]              % DIM as normal

"check" [ % offset sizenum namestring -> offset sizenum
#2121                   % working copies of offset and sizenum
#11 #+                  %from sizenum ...
1 #+                    % calculate max byte element
#212 #<                 % less than supplied element?
#12 SIGN                % negative?
OR IF( "!! bad element ("$OUT
   #12 NOUT             % print element no.
   ") in "$OUT
   $OUT STOP            % print supplied name
   )IF
$2 ]                    % discard supplied name
"chars" [
4
"chars” check
DIM 1 #+ ]

"$!" [ % string addressnumber $!
LEN #212 #B!            % store length
LEN FOR(                % for each character...
  1 #+                  % next location
  1 $- ASC #212 #B!     % store character, leaving remainder
)FOR
$2 #2 ]                 % discard null string and address
"$?" [ % addressnumber $? -> string
""
#11 #B?                 % fetch length
FOR( 1 #+               % next location
  #11 #B?               % get character
 $CHR $+                % add to string start
)FOR
#2                      % drop address
$REV ]                  % turn string right way around
...
"$var" [ 63 DIM ]       % define string variable
...
$IN $ var $!            % store (inside [])
...
$var $? $OUT            % fetch (inside [])
```

related words ARRAY #! #? #+! #B! #B?

further information

The first time the DIM instruction is executed, it finds and reserves the memory, and subsequently it ignores the size number. If the size is less than zero, a value of zero is used. The

maximum size number allowed is 16383.

DIM takes memory from the program area, and its consumption is shown by MEM. The record of DIM memory is cleared by any command that rearranges memory space, including any that moves or removes user words. The first subsequent use of a DIM instruction reserves memory afresh, not necessarily at the same address.

The structure of the user-accessible part of the memory block is as follows:

```
Number of elements in array : element 1 : element 2 ...
|                             |           |
At address-2                  At address  At address+2
```

DIM consumes (sizenumber+5)*2 bytes of memory – this includes 8 bytes of system information.

Where a DIK block is accessed from only one point in the program, the DIM instruction may be used in-line, unnamed. The following example is a word to open a sequential file, using an in-line DIM to provide a control block holding the file name:

```
"fopen" [ filenamestring opnumber fopen -> channelnumber
10 DIM                     % 22-byte block
#11                        %
LEN                        % string length
21 MIN                     % limited to 21 as precaution
FOR(
 1 $- ASC #212 #B!         % move character to buffer
 1 #+ )FOR                 % increment address
$2                         % discard remainder string
13 #12 #B!                 % store carriage return at end
#12                        % leaves: blockaddress opnumber
&FFCE                      % OSFIND entry address
% YX(blockaddress) A(opnumber) entryaddress
CODE                       % leaves: YX PA
#12 #2                     % leaves PA
&FF AND ]                  % leaves A (channel number)
...
"infile" &40 fopen ...     % open for input (inside [])
...
"outfile" &80 fopen ...    % open for output (inside [])
...
"temfile" &CO fopen ...    % open for update (inside [])
```

DISPLAY display text

DISPLAY prints the following lines of text on the screen. The text lines must start with % (making them into comments), and DISPLAY stops at the first line not starting with %, or at the end of the word.

DISPLAY is used as a convenient method to print fixed text, for example, the title of a piece, or a page of instructions.

related words % $OUT

examples

```
"title" [ DISPLAY

%     'Shards'
%        by
%    John Favero
%   November '87
%
]

"page1" [ DISPLAY
%
 ... text for page 1 ...
%
page2 ]
```

DURATION wait for a period of time

```
number DURATION
```

DURATION makes the specified number of timebase ticks elapse before the player's sound continues. It is used to make a sound play for a certain period of time by delaying the onset of the next sound.

Music event words generate their own duration (from their length setting), so DURATION is usually confined to additional effects in music notation, such as extra delays that do not contribute to bar length, and use in programs which employ sound words directly, including music action definitions.

The number must be in the range -32768 to 32767. With the normal timebase period, each unit corresponds to 10 milliseconds. =T allows the actual duration of the tick to be changed.

examples

```
   "fermata" [ 80 DURATION ] wait for 80 ticks
   "part1" [
   SCORE 24, 3 BAR
   ...
   12,ADf/ed |
   D///// fermata |  % hold last note (with bar length as normal)
   0,^ ]

   "part1" [ SCORE 48, 4 BAR  % bass drum
   4 FOR(
    8 FOR( XXXX | XXXX | )FOR % crotchet beat for reference
   )FOR ]
   "part2" [ SCORE 48, 4 BAR  % snare drum
   3 DURATION                 % put behind (after) the beat
   8 FOR( /X/X | /X/X | )FOR
   -3 DURATION                % put back on the beat
   8 FOR( /X/X | /X/X | )FOR
   -3 DURATION                % put in front of (before) the beat
   8 FOR( /X/X | /X/X | )FOR
   3 DURATION                 % put back on the beat
   8 FOR( /X/X | /X/X | )FOR ]
   ...                        % play part1 & part2 together

   "metronome" [ % strike (existing) voice every 20 ticks
   REP(
     1 VOICE QN GATE          % strike
     20 DURATION              % wait 20 before next
   )REP ]

   "talkback" [
   READY 1 VOICES ins         % 'ins' to suit installation
   REP(
    16 FOR(
     12 RANDL 24 #+  PITCH ON GATE
     10 DURATION              % repeated short
    )FOR
    OFF GATE
    100 DURATION              % single long
    4 FOR(
     12 RANDL 24 1-  PITCH ON GATE
     40 DURATION              % repeated medium
    )FOR
    OFF GATE
    20 DURATION               % single short
   )REP ]
```

related words QTIME =T

further information

DURATION adds the given number to the player's program time, moving forwards or backwards, possibly passing over sound events issued previously. The practical limit on backwards movement is the total queue time itself, returned by QTIME. Under normal conditions, the queue time before reduction by negative durations is determined by the total queue capacity, which is 220 events for all players.

A player's time begins to pass at the instant its P sequence begins or in the case of player 0, when the command line begins execution, and continues regardless of whether durations are sent. If the player has sent nothing for a long time, durations will be consumed rapidly to make up this lost time, ensuring that the temporary hold-up does not cause a permanent error and loss of synchronisation with other players.

In some situations, the player needs to start time afresh after sending no durations for a time. An example is a program that waits until a key press before playing a short tune. To ensure that the tune starts on time, the player sends a duration equal to the time that has passed,. as follows:

```
QTIME           % read queue time – should be negative
0 #l2 #-        % negate
DURATION        % send at duration to make QTIME up to zero
```

Here is an improved version that also will not cause overlap on an unfinished previous tune:

```
"makeup" [ % make-up lost time to start afresh
0 QTIME #-
0 MAX
DURATION ]
```

It is used before the start of the tune, for example

```
"keyplay" [
REP( #IN #2               % wait for key
makeup                    % make up lost time
SCORE 12,gABCDcbag^       % play tune
)REP
```

The contrasting type of program in which occasional user input affects music that runs continuously, can also use QTIME – see QTIME for details.

)ELSE(separate conditional sections [] only

)ELSE(is used inside IF(...)IF to introduce a sequence of words that is done if the tested flag is OFF.

See IF(for more information.

EVERY leave 'every' selector

EVERY is a constant for use with various selecting words, where it selects every one of the items together. For example, EVERY VOICE selects all voices in the current voice range, set by VOICES.

EVERY leaves the value -1, and is equivalent to ON.

related words VOICE ; ON

FAST select fast/normal tempo

flag FAST

FAST controls whether musical time passes normally, or runs as fast as the program will go. ON FAST selects fast execution, and OFF FAST returns to normal.

FAST can be used to skip over sections of music, or run through the whole piece at top speed to test for errors such as 'Bad bar'.

further information

Unlike WIND, ON FAST makes time pass only as fast as the program can run, ensuring that the players remain in synchronisation while running fast.

FCOPY copy numbers from frame pointer

number -> number 1 ... number-n

FCOPY copies the specified number of numbers from underneath the FRAME pointer to the top of the stack, preserving the order. It is used to access groups of numbers without destroying them. FCOPY is much faster than the equivalent FOR/FVAR instruction sequence.

examples

```
3 FCOPY     % copy 3 numbers from FRAME

2 4 6 FRAME % leave three numbers, marked with FRAME
```

```
   8 10        % add some more on top
   3 FCOPY     % copy 3 to top
   NOUT        % prints  6
   NOUT        % prints  4
   NOUT        % prints  2
               % leaves: 2 4 6 8 10

   "dump" [ % number dump ->
               % print top n numbers (not inc n!), going down
   FRAME                     % mark
   #11 1 #+ FCOPY            % copy n numbers AND n itself
   FOR( NOUT SP )FOR         % print n numbers
   #2 ]                      % discard n
   ...
   1 1 2 3 5 8  6 dump       % prints  8 5 3 2 1 1
                             % leaves: 1 1 2 3 5 8
```

related words FRAME

FIND find uses of word command

namestring FIND

FIND displays the names of all words which use the specified word. The specified word can be a system or user word, and module words with readable definitions also searched. FIND is useful for finding where a particular user word is used in the program, and displaying a list of all words of a certain type, for example all instruments, by searching for a key word.

The list of finds is given in groups: program and each module.

```
examples       "riff1"FIND     % find all uses of riff1
               "SCORE"FIND     % find all uses of SCORE
```

related words SHOW

FOR(start definite loop [] only

countnumber FOR(...)FOR

FOR(...)FOR carries out the instructions inside it the specified number of times. It is particularly used to repeat musical sections.

If the number of repeats is less than one, the contents are not executed at all. FOR(...)FOR can only be used inside words.

The words COUNT and INDEX give the number of executions done and the number to be done, respectively.

examples

```
"stars" [ % number stars
FOR( "*" $OUT )FOR ]
5 stars                  % prints   *****
0 stars                  % prints nothing

"pnout" [ % number pnout % prints number in field of 8 stars
$STR                     % convert number to dec string form
8 LEN #-                 % calculate how many stars needed
FOH( "¿" $+ )FOR         % add them to start
$OUT ]                   % print
42 pnout                 % prints   ******42

phrase1                  % (in a word definition)
8 FOR( phrase2 )FOR      % phrase2 is done 8 times
phrase3

SCORE -2: 8,             % (in a word definition)
4 FOR( CDEFGAB )FOR ^ ]  % play a scale over four octaves

"tri" [ % print triangular area
10 FOR(                  % on lines 1 to 10...
 COUNT FOR( "*"$OUT )FOR % print 1 to 10 stars
 NL                      % next line
)FOR ]
tri
```

related words COUNT INDEX REP(

)FOR end definite loop [] only

FOR(...)FOR carries out the instructions inside it the specified number of times.

See FOR(for more information.

FRAME set fr ame pointer to top of stack

FRAME marks the current top of the number stack as the start of a stack frame which can then be accessed like an array, using FVAR. FRAME enables direct access to numbers on the stack, so that the stack can more easily be used for temporary storage of more than just a few numbers.

The current FRAME value can be saved and restored with FRAME? and FRAME!, allowing outer and inner levels of a program to use FRAME

independently. Music events use FRAME to mark music event values on the stack, and therefore the FRAME value is undefined over music events, so you should save and restore it around music events if the calling word needs it to be preserved.

examples See FVAR

related words FVAR FRAME? FRAME! FCOPY

FRAME! write frame pointer

pointernumber FRAME!

FRAME! sets the stack frame pointer to the value supplied. It is used to restore the pointer to the value read with FRAME?.

examples See FVAR

related words FRAME FRAME? FVAR

FRAME? read frame pointer

FRAME? -> pointernumber

FRAME? reads the value of the stack frame pointer. It is used to save the pointer value on the stack for later restoration by FRAME! .

examples See FVAR

related words FRAME FRAME! FVAR

FVAR access stack frame item

elementnumber FVAR -> addressnumber

FVAR is used to access the items in the stack frame marked by FRAME as elements of an array. It is used for convenient access to temporary values on the stack, and to access temporary var iable storage on the stack •

It takes the element number and returns the address of it, for use by #?, #!, #+!, #B? and #B! . The position of the top of the stack when FRAME was used is element number 1.

examples

```
FRAME 4 FVAR #?     % #43214 – copy the fourth item to the top
```

```
   FRAME               % non-destructive stack print
   3 FOR( COUNT FVAR #?
   NOUT SP             % print the top 3 items
   )FOR                % leaving them unchanged

                       % method of random access to stack location
   0                   % leave one number on stack
   FRAME               % mark it for later access
   ...                 % other operations, putting items on stack
   1 FVAR #!           % store value to variable
   ...                 % other operations
   1 FVAR #?           % read value from variable

   ",?" [ % ,? -> number  % return ',' setting, from MVAL?
   0                   % dummy item for storage
   MVAL?               % seven items, including ',' as no. 2
   FRAME               % mark
   2 FVAR #?           % get no. 2 out
   8 FVAR 4!           % and store over dummy item
   MVAL!               % get rid of the MVAL? numbers
   ]                   % leaving result on stack
   ...
   48, ,? NOUT         % prints  48

   "pitch"  [ 1 FVAR ] % define named action variables
   "pitchv" [ 2 FVAR ] % for clearer action definitions
   "level"  [ 3 FVAR ] % (see ACT(
   ...
   "dur"    [ 7 FVAR ]
```

related words FRAME FRAME! FRAME?

GO start players together

GO starts the players executing together, after allowing them time to prepare to play. It is only used in combination with READY and a list of P(...)P structures, usually in the 'RUN' word of a piece.

GO cannot be used inside a player, and is used only once after READY. Normally, there should be no DURATIONs (or music events, which themselves use DURATION) between READY and GO.

example

```
"RUN" [
READY mix
1 P( part1 )P
2 P( part2 )P
GO ]
```

related words READY P(GATE DURATION QTIME

further information

GO allows players to continue when every one of them has sent a GATE, signalling that the first note is ready to play, though players that send no sound events at all are exempt. In special applications where a player does not send a first note, it should send a dummy GATE on voice 0 to signal that it is ready:

```
OFF VOICE OFF GATE
```

GO's sequence of operations is as follows:

1 The timebase is halted
2 Players are executed, using IDLE, until either the sound event queue is full, or no more sound events are issued. At this time, the system executes each player's sound events up until its first GATE event (voice selection immaterial), including for example initial voice assignment events, but holds the GATE and subsequent events on the queue.
4 The timebase is allowed to continue, and execution of held events begins.

GVAR create variable [] only

GVAR -> addressnumber

GVAR creates a number variable. It is used inside [...], which gives the variable its name. #? and #! are used to fetch from (read) and store to (set) the variable.

```
store to variable:          number variable #!
fetch from variable:        variable #? -> number
```

Variables are not used as much in AMPLE as in other languages, because AMPLE's stack can be used for temporary storage and passing values to and from words. Before deciding to use a variable, you should think about whether it would be simpler to use the stack.

GVAR variables are global to all players, so a value stored by one player can be read by any other . The initial value is undefined.

examples

```
"var" [GVAR]              % create variable 'var'
0 var #!                  % set to0
var #? NOUT               % print value
```

```
   var1 #? var2 #!           % var2 := var1
   var1 #? 1 #+ var2 #!      % var2 := var1 + 1

   varl #? var2 #?           % swap var1 and var2
   var1 #! var2 #!

   % use of variable to engage chain of next prog at end of piece
   "var" [GVAR]             % variable to hold message
   "speak" [                % sends 'end reached' message
   REP(
    QTIME -100 #< )UNTIL(   % wait for 100 ticks past last event
    IDLE )REP               % (or longer if long fade past end)
   ON var #! ]              % set var to ON
   "listen" [               % wait for ON message
   REP( var #? )UNTIL(      % repeat until var has been set to ON
    IDLE )REP ]             % by another player
   "part1" [                % part1 is longest part
   OFF var #!               % initialise message variable
   ...                      % play music
   speak ]                  % send 'end reached' message
   ...                      % rest of program
   "RUN" [                  % main word
   ...                      % start piece playing
   listen                   % wait for 'end reached' message
   $2                       % discard existing command line
   """nextprg""LOAD RUN" ]  % leave command to chain next program
```

related words #? #! #+! DIM ARRAY

further information

The GVAR instruction itself includes the storage space for the variable.

HALT halt/continue timebase

flag HALT

HALT controls the timebase:

```
   ON HALT     stops timebase
   OFF HALT    allows timebase to continue
```

While the timebase is stopped, all durations last indefinitely so that music is frozen. HALT is used to temporarily halt the music while it is playing. While halted, the music can still be advanced by WIND.

The timebase is automatically allowed to continue by READY and when an error occurs.

examples

```
% function keys to hold and resume music
*KEY4 ON HALT|M         % halt music
*KEY5 OFF HALT|M        % continue music
*KEY6 192 WIND|M        % wind music while halted
```

further information

The main difference between HALT and PAUSE is that HALT just stops the passage of time, whereas PAUSE also stops the execution of sound messages.

#IN wait for and get keypress

```
#IN -> asciinumber
```

#IN waits for a character from the keyboard and returns its code.

If there is already a character in the keyboard buffer when #IN is called, it returns the character immediately.

example

```
% wait for RETURN press
"RETget" [ REP( #IN 13 #= )UNTIL( )REP ]
```

related words #OUT QKEY

further information

#IN calls 0 QKEY so that it can IDLE until a keypress is available. Calling OSRDCH would halt other players.

$IN input line from keyboard [] only

```
$IN -> str ing
```

$IN accepts a line of characters from the keyboard, terminated by RETURN or TAB. The terminating character is included at the end of the string.

$IN lets the program accept a line of characters from the user. This can be used as a string, or converted into a number using VAL or &VAL. Leading spaces are included, but they can be removed with $STRIP.

The special key functions of $IN are the same as those of the system's % command line. The DELETE key removes the last character, and CTRL-U discards the line but leaves it on the screen. No control codes (apart from CR and TAB) are included in the input line. All control codes except 22 (select screen mode)

are sent to the screen via #OUT.

examples

```
$IN ASC             % alternative to kIN, but waits for RETURN

% input number: nin -> number ON
                    or  nin -> OFF if no number found
"nin" [
$IN                 % get string
$STRIP VAL          % convert to number
$2 ]                % discard remainder of string

% alternative input number: ninz -> number (0 if no number)
"ninz" [
$IN                 % get string
$STRIP VAL          % convert to number
$2                  % discard remainder of string
NOT IF( 0 )IF ]     % leave 0 if no number detected by VAL
```

related words #IN $STRIP VAL &VAL

further information

$IN bleeps if you try to add another character when the line is at maximum length. $IN will accept a line right up to the maximum length that can be accommodated on the string stack, but because there is a total limit (of 128 characters), this length will be decreased by other strings on the stack.

$IN resets various keyboard and screen options before accepting input, as follows:

```
*FX255,1                    % make function keys expand
*FX4,0                      % engage cursor editing mode
OSWRCH: 23 1 0 0 0 0 0 0 0  % turn cursor on
```

IDLE pass control to other players

IDLE passes control to other players allowing them to continue execution.

IDLE is used in loops that wait for an external event before continuing, so that other players are not held up. IDLE is not normally required in loops that issue sound or music events.

example

```
% wait for CTRL key to be down
"CTRLwait" [ REP( -2 QKEY )UNTIL( IDLE )REP ]
```

further information

Some Nucleus words can wait for an external event before returning, and they have the effect of IDLE while waiting. These include:

```
#IN, $IN
A-G, a-g, X, /, ^, (, ) (all music events)
ACT, DURATION, HALT, FAST, ON PAUSE
+T, -T, =T
P(
sound words (PITCH, GATE etc)
some commands, including WRITE
```

IF(start conditional sequence [] only

flag IF(...)IF or flag IF(...)ELSE(...)IF

IF(...)IF carries out the instructions inside only if the flag is ON. IF(...)IF is used to carry out operations or not depending on the results of previous calculations.
An)ELSE(can be included between IF(and)IF. In this case, if the flag was off, the instructions after)ELSE(are carried out.

... flag IF(.. done if ON ..)ELSE(.. done if OFF ..)IF ...

IF structures can only be used inside word definitions.

example

```
"test" [ IF( "ON" )ELSE( "OFF" ) IF $OUT ]
                    % ON test prints ON
                    % OFF test prints OFF
```

further information

IF(treats all non-zero values as ON, so you can use it directly to test a number for non-equality to zero.

)IF end conditional [] only

IF(...)IF and IF(...)ELSE(...)IF enclose words which are executed conditionally.

See IF(for more information.

INDEX leave loop index [] only

INDEX -> number

INDEX leaves the index of the most recent FOR(...)FOR loop containing it. The index starts at the maximum (the loop count given to FOR( ) and decreases by one each time around the loop. On the last time around, it is one.

INDEX lets the instructions inside the loop do something different on each pass, usually sequencing through a particular range of values. In many cases, COUNT is more convenient.

The INDEX can be put anywhere inside the FOR (...) FOR loop, except enclosed in a separate wor d ([...]) or intervening P(...) P structure.

example

```
"countdown" [  % print numbers from 20 down to 1
20 FOR( INDEX NOUT SP )FOR NL ]
```

related words FOR(COUNT

further information

The functions of INDEX and COUNT differ only in direction of counting. INDEX is slightly faster in execution than COUNT.

INSTALL install module command

namestring INSTALL

INSTALL installs the named module as 'fixed'. All modules already present in memory also become fixed.

example "INT" INSTALL

related words MCAT MDELETE MLOAD MPREFIX

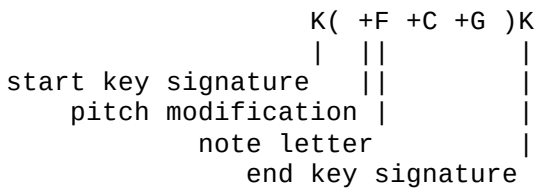
further information

INSTALL stops players 1-10 and all sounds, and memory is compacted before the new module is installed. Any module can be INSTALLed but then cannot be disposed of with MDELETE. Some modules may only be installed, and will cause a 'Fixed only' error if an attempt is made to MLOAD them. INSTALL is usually used from a !BOOT file.

K(start key signature

K(...)K sets the key signature for the player. A key signature is a set of sharps and flats for particular note letters which are automatically applied to each of those note letters in future.

To set the key signature, enclose a list of the note letters (upper or lower case), each with a sharp or flat before it, inside K(...)K:



Where is single key signature is used throughout a piece, it is sometimes set out in a single 'sig' word which also includes SCORE and BAR instructions, and is then called up by all players. Alternatively, the global key signature may be stated after READY in the form of RUN word which uses a list of P(...)P structures – see P(for details.

An individual note can be returned to its unmodified pitch by putting a natural symbol, '=', before it. This effect applies to that note only. The sharp and flat symbols (+ – or =) also cancel the effect of the key signature for the next note.

Chord brackets and more key signatures are not allowed inside key signatures. Both upper-case and lower-case note letters may be used inside K(...)K, with the same effect.

examples

```
K( )K               % C major
K( +F +C +G )K      % A major
K( -B )K            % F major

"amaj" [ K( +F +C +G )K ]

"sig" [             % complete signature
SCORE K( -B )K
48, 4 BAR ]

READY K( -B )K      % key signature for all parts (see READY)
... rest of RUN ...
```

related words + – =

further information

The key signature can be freely changed in the middle of music.

There is no restriction on the modifications inside key signatures, so non-standard key signatures can be created. This is particularly useful for minor keys, since the raised 7th can be included, for example, D minor: K(-B +C)K.

Notes inside key signatures do not play or alter the effective previous note pitch.

)K end key signature

K(...)K sets the key signature for the player.

See K(for more information.

LEN get length or string [] only

string LEN -> string lengthnumber

LEN returns the length of the string (the number of characters in it). It leaves the string on the stack as it found it.

example LEN 0 #= % test if string is null (inside word)

LOAD load program command

namestring LOAD

LOAD loads the named program file. The existing program is entirely replaced by the new one.

examples "myprog" LOAD
"myprog" LOAD RUN % (RUN is user word)

related words SAVE MERGE

further information

LOAD stops players 1-10 and all sounds, and resets player 0's music action chain.

'L set accent strength

'L sets the amount of dynamic level added by ' (accent), that is, the strength of accents. The range is -127 to 127.

SCORE sets the accent level to 15.

Remember that the available range of dynamic level depends on the type of voice in use.

example

```
SCORE 30'L      % make accents stronger

-20 1L          % set reverse (quieter ) accents
```

related words ' = L

=L set dynamic level

number = L

=L sets the dynamic level of hits and notes in the range 0 (soft) to 127 (loud). 0-127 is the maximum range possible, but a particular voice type may respond to only part of this range.

64 is the normal value, set by SCORE.

=L cancels any +L or -L level change in progress.

example

```
80 =L                       % set medium loud level

64=L 40 16 +L               % set level before crescendo

"ppl" [ 24  =L ]          % define words for set dynamic markings
"pl"  [ 44  =L ]
"ml"  [ 64  =L ]
"fl"  [ 84  =L ]
"fff" [ 104 =L ]
```

related words +L -L

further information

Remember that the effect of the level depends on the voice type in use, and possibly the instrument also. Also note that the dynamic level only takes effect at the start of a note or hit, not within it.

+L increase dynamic level

changenumber eventsnumber +L

+L increases the dynamic level of the player's voices by a specified amount over a specified period of time. It is used for making changes of level relative to the current setting, both instantly and automatically over a period of time (crescendo).

The first number is the amount of change in the dynamic level, and has the range -127 to 127. A positive value gives an increase, and a negative value gives a decrease. The second number is the number of events, of the length setting in use at the time of the +L, over which the change is to take place. These events can be notes, rests, hits and ties, and the length setting can be changed after the +L without affecting the length of the change period. A value of zero makes the change happen instantly. At the end of the change period, the level is left at the final value.

The change follows a slope starting at the +L and ending at the end of the last event. This means that the first note is unaffected (unless the change is instant), and the first note that plays at the final level is the one after the last event on the slope. In practice, this is the effect you would expect.

Remember that the range of the dynamic level itself is 0 to 127, and that the effect of the level depends on the instrument in use. Also note that the dynamic level only takes effect at the start of a note, not within it. A crescendo within a note is a feature of the instrument, and must be created by programming an envelope.

examples

```
48, 10 4 +L        % crescendo of 10 units over four crotchets

12, 40 8 +L XXXX XXXX X     % drum roll with crescendo

48, 40 2 +L 12,XXXX XXXX X  % equivalent

60 =L   CDEG       % a bar with a dynamic level of 60
40 4 +L GABC       % then increasing to 100 over a bar
40 4 -L Dcba g     % then decreasing back to 60 over a bar

4, 40 4 +L 48,D(4,FACD)     % crescendo over broken chord

"at" [             % user-defined accent symbol
20 0 +L            % instant increase by 20 units
20 1 -L            % decrease by 20 units after next event
]
...
```

```
   at Cggg at Cggg    % accent both Cs
   at E(GB)/          % accent whole chord
```

related words =L -L

further information

The change is halted by another +L, -L, =L or SCORE instruction. 0 0 +L can be used to halt a change, leaving the level constant at whatever value it had reached.

+L cannot be used with a ',' setting of zero (in normal chords, for example). Attempting to do so gives the 'Division by zero' error .

Though the effective range of the level (instantaneous =L value) is 0 to 127, the stored value is held accurately up to 255. This means that even if a relative change exceeds the maximum, an opposite change will still return to the starting point. However, there is no such margin at the bottom of the range, and the level is clipped at 0.

-L decrease dynamic level

```
changenumber eventsnumber -L
```

-L decreases the dynamic level by a specified amount over a specified per iod of time. It is used for making changes of level relative to the current setting, both instantly and automatically over a period of time (diminuendo).

-L is exactly equivalent to +L, except that positive change values give a decrease (and negative change values give an increase). See +L.

MAX leave largest of two numbers

```
   number1 number2 -> largestnumber
```

MAX leaves the largest of the two numbers and discards the other. It is often used to make sure a variable value does not go below a fixed limit, enforcing a minimum value.

example -5 -2 MAX produces -2

related words MIN

MCAT display catalogue of modules command

MCAT lists the names of all the modules in the system in alphabetical order, together with their version numbers and status indicators. The indicators have the following meanings:

Indicator	Meaning
F	Fixed – fixed modules are loaded using INSTALL by the system disc start-up. Fixed modules are part of the installation and cannot be removed by MDELETE. The command AMPLE leaves fixed modules installed.
P	Program-owned – the module was loaded automatically on load of a program that required it. P modules are removed automatically when the program is removed, and may be deleted with MDELETE if not in use.
T	Tempora y – the module was loaded manually by MLOAD, and may be removed at any time (provided not in use) by MDELETE. The command AHFLE removes temporary modules.
U	In use as editor – the module is the currently in-use editor. The module may not be removed by MDELETE while it is in use as the current editor.
W	In use by words – one or more words of the module are in use by the user program. The module may not be removed while words are in use. FIND can be used to locate the uses of the words.
M	In use by modules – one or more words of the module are in use by other modules. The module may not be removed while words are in use.

The F, P and T statuses are mutually exclusive.

example

```
          %MCAT
         INT 0.2 F       PAD 0.2 TU      M5 0.7 FW
          |   |  |               ||              |
Module name   |  |               ||              |
 Version number   |               ||              |
      Fixed module                ||              |
                  Temporary module |  words in use
                   in use as editor    by program
```

related words INSTALL MDELETE MLOAD MSHOW

MDELETE delete module command

namestring MDELETE

MDELETE removes the named module. This command is used to remove modules that are no longer required, freeing their space for other uses.

Note that the module name must be given in the correct case – module names are usually all upper case.

The module may not be removed if it is fixed or in use, that is, if its MCAT display shows any of the indicators F, U, W or M.

examples "PAD" MDELETE

related words INSTALL MCAT MLOAD

further information

Some special modules load further modules when loaded. When deleting such a module with MDELETE, you will also need to delete the further modules manually, using the MCAT display as a guide.

MEM show memory usage in bytes command

MEM shows the number of bytes of user memory in use for various functions.

The following figures are given:

Words	memory used by the program, that is, word definitions, not including editor data
Data	memory used by editors for public data
System	memory used by players and music actions
Arrays	memory reserved by the program with DIM
Free	memory free for use

Note that the free memory may be fragmented so that the largest single piece may be smaller than the total figure given. See COMPACT.

example

```
%MEM
Words: 2569      Data: 234
System: 972      Arrays: 0
Free: 12330
```

further information

The total amount of user memory depends on the value of OSHWM (determined by ROMs fitted to the machine), and, on a BBC Microcomputer without shadow RAM, the screen mode in use. Without shadow RAM, mode 7 gives the maximum amount of language memory.

The total amount of program memory is the amount of language memory minus that used by modules.

MIN leave smallest of two numbers

number1 number2 -> smallestnumber

MIN leaves the smallest of the two numbers and discards the other. It is often used to make sure a variable value does not go above a fixed limit, enforcing a maximum value.

example 4 5 MIN produces 4

related words MAX

MLOAD load module command

namestring MLOAD

MLOAD loads the specified module. The module filename prefix set with MPREFIX is added to the start of the filename.

The loaded module is temporary (shows with indicator T in the MCAT display) and can removed with MDELETE when no longer required.

MLOAD prints messsages indicating the full prefixed names of the modules it loads.

examples "LEDIT" MLOAD

related words INSTALL MCAT MDELETE MPREFIX MSHOW

further information

The disadvantage of using MLOAD over INSTALL is that all but the first module loaded with MLOAD will take up slightly more memory than otherwise. In normal use, only one temporary module is loaded at a time, so this has no effect.

Some special modules require further modules, and these will be loaded automatically. MLOAD prints messsages indicating the prefixed names of the original and any further modules it loads.

MODE enter display mode command number MODE

MODE enters the specified display mode.

On BBC Microcomputers without shadow RAM, different display modes use different amounts of memory, and you will normally use mode 7 since this uses least memory.

If the memory required by the new mode is not available, you will get the 'No room' error. In this case, you may still be able to change to the new mode after entering COMPACT.

example 3 MODE % change to mode 3

related words #OUT COMPACT

further information

MODE can be freely used in programs. Since programs are saved in compacted form, LOAD RUN will never give a needless 'No room' error.

MPREFIX set module filename prefix command string MPREFIX

MPREFIX sets the filename prefix to be used by MLOAD and INSTALL to load modules. It allows modules to be held in and loaded from specific directories and drives. The prefix is set to null on *AMPLE (start-up) and AMPLE (restart).

Modules are usually supplied under directory 'M', so the system disc sets "M."MPREFIX on booting.

examples ":0.M" MPREFIX % load modules from drive 0

related words INSTALL MLOAD

further information

The maximum prefix length is 9 characters.

MSHOW show list of words in module command

modnamestring MSHOW

MSHOW displays the names of all the words in the specified module. If the module contains auxilliary command words, as do editors, they appear in a separate list.

Note that the module name is normally in upper case.

example

```
 %"PAD" MSHOW
PAD
Aux:      CLEAR     GET      MAKE
NAME      PANEL     RETGATE  TRY
 %
```

related words SHOW MCAT

MVAL! write music variables

framelev keysig barcountlen octnote length tranvoice MVAL!

MVAL! writes the important music variables. The input values are the same as the MVAL? output values. See MVAL? for more information.

related words MVAL?

MVAL? read music variables

MVAL? -> framelev keysig barcountlen octnote length tranvoice

MVAL? reads the important music variables – the 'music environment' values used by the 'music event' words. It copies them from working locations to a byte-packed form as seven numbers on the stack.

MVAL? is supplied for advanced programs which need to access the music settings for special processing, extended music functions etc. It can also be used with MVAL! to save and restore the music variables around 'local' music sections so that their music settings do not affect the settings in the sections that contain them.

name	position	low byte	high byte
tranvoice	1 (top)	voice (;)	transposition (@)
length (,)	2	low	high
octnote	3	effective last note	octave (:)
barlen (BAR)	4	low	high
key signature	5	sharps	flats
framelev	6 (bottom)	dynamic level (=L)	FRAME pointer
barlen count	7	low	high

The 'effective last note' is the combined record case and letter used for the determination of the relative pitch octave of the next note:

```
bits 0-2    letter number: 0 = C, 1 = D, ... 6 = B
bit 7       case: 0 = lower case, 1 = upper case
```

The 'octave' value is a semitone pitch offset from middle C which records the curr ent pitch octave, always being a multiple of 12. It is set by ':' and incremeted or decr emented by note letters.

The two key signature bytes record the state of sharp and flat modification of each of the note letters, one bit per letter:

```
bit 0       C
bit 1       D
...         ...
bit 6       B

bit = 0     normal
bit = 1     sharp or flat
```

The FRAME value is just as returned by FRAME?. Though the number stack is not used for storage of the music environment values, it is used to hold the music action values. These are marked by FRAME, for access inside ACT(...)ACT. This means that FRAME must be preserved through the ACT(...)ACT contents, so if further music events are included inside ACT(...)ACT itself, it must be saved and restored. Because FRAME is MVAL, is is saved and restored along with the music environment values.

examples

```
MVAL?       % save music variables
2; 4@
3: CDEG
MVAL!   % restore to state before MVAL?

",?" [      % return current length setting (item 2)
MVAL?
```

```
#2          % discard item 1
5 FOR( #12 #2 )FOR % discard items 3 to 7
]

"riffact" [ % 'note expansion'
1 ACT(
MVAL?          % save music values
2 FVAR #?   % get pitch
@           % transpose to this
12,0:CCGG   % play transposed riff
MVAL!               % restore music values
)ACT ]
...
SCORE riffact       % engage note expansion
0: C D              % plays 0:CCGG ddAA
ccccFFGG ccccb+aa+g gggg
```

related words MVAL!

NEW discard program command

NEW discards all user words, ready for a new program to be entered. Editor data (text) is not affected. NEW should be used with care as its effect cannot be reversed.

To do a complete clear befor e entering a new program, use the command AMPLE.

related words AMPLE CLEAR

further information

NEW stops players 1-10 and all sounds, and resets player 0's music action chain.

NL print new line

NL moves the cursor to the start of the next line, ready for printing on a new line.

example

```
"     Elegy"$OUT NL NL " by John Favero"$OUT NL
```

prints the following:

```
     Elegy

 by John Favero
```

related words ALIGN $OUT #OUT

further information

NL calls OSNEWL.

NOT invert sense of flag

flagl NOT -> flag2

NOT inverts the sense of the flag, that is, it replaces ON by OFF, and OFF by ON. Is it used in logical expressions, and often before IF(...)IF so the contents are carried out if the flag is OFF.

examples

```
#< NOT  % ON if number was greater than or
        % equal to previous number

$IN VAL NOT IF( 0 )IF % leave number or 0 if none
```

related words AND OR XOR

further information

NOT is not a bitwise operator like AND, OR and XOR. It treats any non-zero number as ON. For a bitwise NOT, use &FFFF XOR.

NOUT print number in decimal

number NOUT

The number is printed on the screen in decimal. It is printed at the cursor position with no formatting spaces.

$&STR can be used to convert the number to the equivalent string for formatting prior to printing.

examples

```
56 NOUT   prints   56

"ppitch" [ "Pitch: " $OUT NOUT NL ]
...
32 ppitch       % prints   Pitch: 32
```

related words &NOUT $STR

further information

For control over printing format which is not provided by NOUT, the user can use $STR plus further string operations – see $STR

for details.

&NOUT print number in hexadecimal

number &NOUT

&NOUT prints the number in unsigned hexadecimal. It is printed at the cursor position without formatting spaces.

$STR can be used to convert the number to a string which can be formatted before printing.

examples

```
255 &NOUT    prints   FF

"pregs" [
NL "PA: &" $OUT &NOUT
" YX: &" $OUT &NOUT ]
...
%FFOO 533CA pregs   % prints   PA: %33CA YX: &FFOO
```

related words NOUT $&STR

For control over printing format which is not provided by &NOUT, the user can use &$STR plus further string operations – see &$STR for details.

OFF leave off flag value

OFF -> offflag

OFF is the 'false' flag constant. It is used with commands and other words that accept a flag, and in logical expressions.

examples

```
OFF PAUSE         % turn PAUSE off
flagvar #? OFF #= % equivalent to   flagvar #? NOT
```

related words ON

further information

OFF is represented by the numeric value 0.

ON leave on flag value

ON -> onflag

ON is the 'true' flag constant. It is used with commands and other words that accept a flag, and in logical expressions.

example

```
ON PAUSE  % turn PAUSE on
```

related words OFF EVERY

further information

ON is represented by the numeric value -1.

OR OR bits of numbers

number1 number2 OR -> ORnumber (number1 OR number2)

OR performs the logical OR operation on the bit patterns of the two numbers. Each bit in the result is 1 if the corresponding bit in either of the input numbers is 1.

OR is used both as a bit-wise operator for manipulating bit patterns, and as a logical operator for flags:

bit1	bit2	bit3	flag1	flag2	flag3
0	0	0	OFF	OFF	OFF
0	1	1	OFF	ON	ON
1	0	1	ON	OFF	ON
1	1	1	ON	ON	ON

examples

```
1 OR                          % set bit 0 of number

#11 -1 #= #12 1 #= OR         % number -> flag
                              % test number was 1 or -1

&1234 &FF OR   produces   &12FF
in binary:
          0001001000110100 OR
          0000000011111111
produces  00010010l1111111
```

related words AND XOR NOT

OSCLI send string to operating system [] only

string OSCLI

OSCLI sends the string to the operating system as a command, as if it had been entered as the keyboard after *. It allows OS commands to be included in programs, and programs to make up OS commands from other data.

examples "CAT" OSCLI

```
"LOAD ""code"" A00" OSCLI

"fkey" [ % string number fkey % defines function key
$12                % swap string to top
" " $+             % add separating space
$STR $+            % add number
"KEY" $+           % add command
OSCLI ]
...
"1 VOICES |M" 9 fkey
```

related words *

further information

OSCLI puts the required carriage return on the end of the string before sending it to the OSCLI routine.

#OUT send ASCII code to screen

number #OUT

#OUT sends the number to the screen. The number may be the ASCII code of a printing character, for example 65 for A, or a control code, for example 10 for line feed.

example

```
12 #OUT            % clear text screen
11 #OUT            % move cursor up one

"cls" [ 12 #OUT ]
```

related words #IN $OUT

further information

#OUT calls OSWRCH.

Never send code 22 to select display mode. Use MODE instead.

$OUT print string [] only

string $OUT

$OUT prints the string on the screen. It is used to display messages from within programs. It can only be used inside a word definition.

$OUT does not print a new line at the end of the string. Use NL for this.

example "hello" $OUT NL % prints hello
% and moves to the next line

related words #OUT $IN DISPLAY

further information

$OUT calls OSASCII.

$PAD pad string to length with spaces [] only

string1 lengthnumber $PAD -> string2

$PAD pads the string to the required length by adding spaces to its start (left end). It is used to make strings appear in fixed columns (fields) when printed.

If the string is not less than the length given, it is left unchanged.

examples "hello" 8 $PAD produces " hello"

```
% print number in field.  number fieldnumber nfout
"nfout" [ #12 $STR $PAD $OUT ]
...
": "$OUT  20 4 nfout   10 4 nfout
% prints   :   20  10
```

related words $STR $&STR $STRIP

P(start concurrent sequence [] only

playernumber P(...) P

P(...)P instructs the numbered player to execute the enclosed sequence of instructions. Program execution continues with the instruction after)P, while the numbered player executes the enclosed sequence alongside at the same time.

P(...)P is commonly used in the 'RUN' word of a multi-part piece, to give the parts of a multi-part piece to the players. The 'RUN' word contains a list of P(...)P structures, one per player, between READY and GO:

```
"RUN" [
READY
...global instructions, such as mix, tempo setting
1 P( ...instructions for part 1... )P
```

```
2 P( ...instructions for part 2... )P
...
n P( ...instructions for part n... )P
GO ]
```

The players are numbered 1 to 10. To refer to a player once the piece has started, to change its sound with SHARE for example, you identify it with this player number.

examples

```
"RUN" [            % piece for two players
READY mix
1 P( part1 )P
2 P( part2 )P
GO ]

"canvas" [               % play background notes for
READY                    % interactive sound editing
1 P( 1 VOICES Simpleins
    SCORE 16,
    REP( -2: 4 FOR( CEGB )FOR )REP
  )P
GO 1 SHARE ]
```

related words READY GO PNUM SHARE

further information

The P(...)P structure is often called a **P structure** and the contents a **P sequence**, for convenience.

A player may receive second and subsequent P sequences from any player including itself. It executes them in the order they were received, becoming idle if it reaches the end of the last one. Player 0 may not be sent P sequences.

Each P sequence synchronises its player to the issuing player, with the result that a sound event issued at the start of the P sequence will play at the same time as one issued at the point of the P structure (that is, immediately before or after). This synchronisation works by P(sending the program time of the issuing player, to be adopted by the destination player before beginning the P sequence. Examples of structures that can be created using synchronised parallel sequences are as follows:

* temporary secondary parts, for example:

  ```
  1 P( ... section1a 2 P( section 2b )P section1b )P
  ```

```
     player 1 plays: <----- section1a -----><---- section1b ----->
     player 2 plays:                        <---- section2b ----->
```

   * chained multi-part sections, for example:

```
       "movA" [ 1 P( sectA1 movB )P
                2 P( sectA2 )P
                3 P( sectA3 )P ] % and similarly for movB and movC
```

   * sequenced multi-part sections, for example:

```
     8 P( SCORE 3072, movA /  movB /  movC / )P % 16 4/4bars
```

READY stops execution of players 1-10 (and discards all voices).

A further feature of the P structure is that each player's first P sequence after READY begins with the music environment, except the music action list, copied from the issuing player. This allows any music environment setting common to all players, such as key signature for example, to be stated once immediately after READY for automatic transferral to every player.

Players 1-10 use some additional memory when in use. MEM displays the total amount in use by players.

GO controls the initial execution of players – see GO for details.

)P end concurrent sequence [] only

P(...)P instructs the numbered player to carry out the instructions inside the brackets, alongside the instructions in any other players.

See P(for more information.

PAUSE pause/continue sound processing

flag PAUSE

PAUSE controls the processing of sounds:

```
   ON PAUSE    stops sound processing
   OFF PAUSE   allows sound processing to continue
```

While sound processing is stopped, the timebase is also stopped, as in the case of HALT. PAUSE is used to temporarily pause the music while it is playing. While paused, no sounds will be processed, even if a FAST is issued.

A special feature of PAUSE is that while the music is paused, any DURATION issued from player 0, for example, / entered as a command, will allow the music to play normally for that period of time, after which it will pause again.

Since PAUSE stops sound processing as well as time, an ON PAUSE immediately before a note, for example, will, as expected, stop that note from sounding. ON HALT in the same position would allow that note to sound, since though it stops time, there is no time interval between it and the sound instructions of the note. Hence, PAUSE is often more useful for pausing at particular locations in music.

Sound processing and timebase are automatically allowed to continue by READY and when an error occurs.

examples

```
RUN        % word to start music playing
ON PAUSE   % pause music
48,////    % allow music to run through 4 beats (only)
OFF PAUSE  % allow music to continue normally

... C  ON PAUSE  D E        % stops before D
... C  ON HALT   D E        % stops on D, that is, before E
```

further information

Since the sound instruction OFF PAUSE is required to execute in the ON PAUSE state, it is a special exception – OFF PAUSE sends a direct message which is acted on immediately, and not queued.

Obviously, it is not useful to include OFF PAUSE from within a sequence of music events or DURATIONs, since it would probably never be reached. OFF PAUSE is normally issued from outside the control of the timebase, that is, in response to an external event such as a direct command from the user.

PNUM leave player number

```
PNUM -> number
```

PNUM provides the number of this player, that is, the number of player executing it. It allows a word or instruction sequence that might be used from any player to find out which player it is being used from, and then make settings or decisions accordingly.

examples

```
   PNUM SHARE     % return this player to using its own voices

                  % automatic progressive delay of canon parts
   "intro" [      % delay intro of successive canon parts
   PNUM 1 #-      % player 1 -> 0, player 2 -> 1 etc.
   FOR(           % repeat <player number – 1> times
    48,////|////| % basic 2-bar delay
   )FOR ]
   ...
   "RUN" [
   READY mix
   3              % number of parts required in canon
   FOR(           % repeat for each part
    COUNT P(      % give instructions to player
    intro part    % after appropriate delay, start common part
    )P )FOR
   GO ]

   "pvar" [       % definition of a player-local variable –
   PNUM           % independent location for each olayer
   10 DIM ARRAY ]
```

related words P(

further information

PNUM returns zero if used from player 0, for example, if the following is entered at the % prompt:

```
   PNUM NOUT
```

QKEY test key status or get keypress

```
   negativenumber QKEY -> flag
       zeronumber QKEY -> asciinumber
```

QKEY performs the 'INKEY' functions: it tests whether a key is down, or gets a character from the keyboard.

A negative number indicates the key to be tested. QKEY gives the answer ON if the key is down and OFF if it is not. See the BBC Microcomputer User Guide under 'INKEY' for a list of the negative key numbers.

Given zero as the number, QKEY returns a keypress from the keyboard. If there is no keypress waiting, QKEY returns a negative number. Programs use 0 QKEY to test for and accept key presses between other operations.

example "shiftstate" [-1 QKEY] % returns state of SHIFT

related words #IN

further information

QKEY calls OSBYTE 129. Positive values should not be used.

QTIME return queue time

QTIME -> number

QTIME returns the current difference, in timebase units, between the player's 'program time' and the system's 'real time', that is, the amount of time by which program execution is ahead of sound execution (the actual sounding of music).

When a player is playing music normally, its QTIME is value is slightly-varying positive. The value of each DURATION it issues is added to QTIME, while QTIME decreases continuously at the timebase (tempo) rate. If QTIME should go negative, it means that the player has failed to deliver music (DURATIONs in particular) as fast as it is being played, and has fallen behind real time.

Uses of QTIME include general timing of external events, and direct control over the differential between program time (accumulated DURATION) and real time (passed time).

Control over the time differential is particularly useful for synchronising sound and non-sound output, for example, music and screen messages - since sound output is strictly 'in-time' and screen output comes directly from program execution, screen output normally appears before any sound output generated at the same point in the program. QTIME can be used to hold back the player's execution until its previous sound has played, keeping the differential near to zero. In time control terms, program time is prevented from running ahead of real time, the queuing of sound messages is disabled and therefore the delay in sound output is eliminated. Under these conditions, the player's sound output will no longer be independent of system load, and, like the screen output, could be held up by any intensive program task, in the same or another player. A solution is to program one player to handle sound output as normal, and a further player to provide the screen output, with minimum time delay, from a non-sound score which simply marks time between screen messages.

examples

```
"keytime" [ % keypress timer – prints time after 2nd press
100, 60 =T                  % 100 ticks/sec & 60 secs/minute
```

```
   REP(
    "Press a key:" $OUT
    #IN #2                      % wait for key and discard code
    NL                          % print new line
    QTIME                       % record time
    "Press a key again:" $OUT
    #IN #2                      % wait for key and discard code
    NL                          % print new line
    QTIME                       % get new time
    #-                          % calculate difference
    "Time interval was "$OUT
    NOUT                        % print it
    "centiseconds" $OUT NL
   )REP ]                       % do it again

   "keymon" [ % continuous keypress timer – prints on each press
   REP(
    #IN #2                      % wait for key and drop code
    QTIME                       % read time
    0 #12 #-                    % negate
    #11 NOUT SP                 % print, but keep
    DURATION                    % return QTIME to zero
   )REP ]

   "swait" [ % waits until last DURATION (or music event) complete
   REP( QTIME 0 #< )UNTIL(     % exit loop when QTIME negative
    IDLE )HEP ]                % (while letting other players run)
   ...
   section1
   swait                        % wait until done
   "Section 1 finished" $OUT    % print announcement
   section2

   REP(                         % wait for 50 ticks after last DUR.
    QTIME -50 #< )UNTIL(        % program time run out by 50 yet?
    IDLE )REP                   % idle and repeat if not
```

related words DURATION

further information

Note that only one player at a time is allowed to leave strings on the stack over IDLE, so for example unless the player 0 is prevented from interpreting input, the following use of the 'swait' word is not allowed:

```
   "Section 1 finished" swait $OUT.
```

Other uses of 'swait' include:

* waiting until the end of the piece, for example to chain another program

* waiting until the last possible moment before issuing a sound that must follow-on from the previous sound, but that is also computed from a real-time user-controlled variable, minimising delay in the effect of the control.

The system command line interpreter takes special action to ensure that a timed sequence, such as a line of music notation, is not consumed by time that passed before the line was entered: if QTIME is less than zero, player 0's program time set so that QTIME returns -1.

QUIT leave editor command

QUIT leaves the current editor. Common uses include exiting the current editor before removing it with MDELETE, and restoring the screen to its normal state after using an editor.

Each command to enter an editor does an automatic QUIT to leave the previous one. The command AMPLE also does QUIT.

RAND get random number

```
RAND -> number
```

RAND produces a random number in the range -32768 to 32767.

You can use RAND to make random decisions and settings in music so it plays differently each time. With RAND! it produces repeatable number sequences which can be used as the basis of computer-generated music.

You will often need to limit the RAND value to a certain range of values before making use of it. An alternative is RANDL which provides numbers in a specified range.

examples

```
RAND NOUT           % print a random number

RAND 0 #<           % leave random flag – ON or OFF

RAND 31 AND         % random number between 0 and 3‘I

RAND &E AND         % even numbers between 0 and 14
```

related words RAND! RANDL

further information

RAND can return 0, unlike random number generators in some other languages. See RAND! for more information.

RAND! set starting point for random numbers

```
number RAND!
```

RAND! sets the random number seed (the number from which the next random number will be generated). For each value set by RAND!, successive calls of RAND and RANDL will generate the same sequence of numbers.

RAND! is used to create repeatable arbitrary number sequences. You use RAND!, preceded by a chosen number, before the start of a loop which contains RAND or RANDL instructions, so that the loop generates the same sequence of numbers every time it is run. Adding a RAND! at the start of any simple program that uses RAND or RANDL will make sure that it gives the same results on each run.

example

```
"rseq" [ % number rtest
RAND!               % print particular sequence
10 FOR(             % of ten numbers
  9 RANDL NOUT SP
  )FOR NL ]
...
0 rseq              % prints a sequence
1 rseq              % prints a different sequence
...
"rrep" [
10 FOR( RAND        % print ten different sequences,
3 FOR(              % each three times
  #11 rseq )FOR
  #2 )FOR ]         % (discard sequence seed)
...
rrep

"randomize" [ % set random number generator by time
              % of key press – different each time
READY
"Press a key, please" $OUT
#IN #2              % wait for key
QTIME RAND! ]       % seed generator
```

related words RAND RANDL

further information

Though RAND! and RAND work on standard 16-bit numbers, the random number generator holds its seed as a 33-bit number – RAND! sets the lower 16-bit directly, and generates the other 17 bits from these while guarding against a zero seed, and RAND returns just the lower 16 bits. Because only part of the seed is accessible by RAND! and RAND, using RAND! with a value taken from an output sequence will not necessarily resume that same sequence.

On system start-up (*AMPLE, BREAK or AMPLE), the seed is undefined, except that it is guaranteed to be non-zero.

RANDL get random number in range

```
maxnumber RANDL -> number
```

RANDL produces a random number between zero and the number given, inclusive. The input number can be positive or negative.

You can use RANDL to make random decisions and settings in music so it plays differently each time. With RAND! it produces repeatable number sequences which can be used as the basis of computer -generated music.

examples

```
9 RANDL NOUT        % print random number between 0 and 9

9 RANDL RANDL       % random number weighted to lower values

18 RANDL 9          % random number between -9 and 9

"rflag" [ % number rflag -> randomflag
RANDL 0 1= ]        % 1/number = probability of ON

"randlb" [ % maxnumber randlb -> randomnumber
            %symmetrical about 0, e.g. -5 <= 5 randlb <= 5
#11
2 #*        % double range
RANDL
#12 #-      % centre on zero
]

"part" [SCORE
REP( 12,
  2 RANDL 1 #-      % choose increment
 10 RANDL 5 #-      % choose starting pitch
 20 RANDL           % choose number of notes
```

```
    FOR(
    #212 #+            % copy and add increment to pitch
    #11 @ 0:C          % play numbered semitone
    )FOR #2 #2         % discard pitch and increment
    300 RANDL          % choose delay
    100 #+ ,/          % (minimum 100)
   )REP                % repeat for ever
   ]
   "RUN" [
   READY
   8 FOR( COUNT        % choices are independent for all 8 parts
     P( 1 VOICES ins   % use a no-sustain, long-decay instrument
        part )P
   )FOR
   GO ]
   RUN                 % play
```

related words RAND RAND!

READY make system ready

READY prepares the system to receive new instructions by making important initial settings of all variables. It is used by the user (as a command at the start of a session) or the program (as an instruction at the start of execution) to make sure that any current activity, such as sounds and players, are terminated, and that the following operations are not affected by previous settings. It does not affect the word definitions or editor state and data.

In a multi-part piece, READY is often used in a 'RUN' word, followed by sound, P(...)P, and a GO instruction – see P(for details of this use.

related words P (GO

further information

READY does the following:

* initialises all voices (silencing all sounds)
* frees all voices from players
* sets 0 VOICES throughout
* sets each player to use the same-numbered ensemble
* stops players 1 to 10, and frees the memory used by them.
* returns PAUSE and HALT to OFF
* sets tempo to 48, 125 =T (125 crotchets/min or 100 ticks/sec)
* discards any sound events and durations waiting to play
* performs a SCORE to reset player 0's music environment values.

Those parameters of a voice type that apply globally to all voices of that type, such as overall tuning, are reset by READY, in this example, to zero. Some voice types may carry out other initialisations on READY.

RENAME rename word command

```
oldnamestring newnamestring RENAME
```

RENAME changes the name of the word to a new name, in the word definition itself and wherever it is used in other words. This command is very useful for changing words to have more meaningful names once a program is complete.

RENAME will not stop you from using a name that is already in use, but warns you that a duplicate name now exists. You can use RENAME again to change it to something else. Until you do, commands will refer to the renamed one rather than the original, so for example you cannot use DELETE on the original.

Note that RENAME does not change any editor text.

examples

```
"temp" "part1" RENAME
"part1" "part1a" RENAME
```

further information

You can use RENAME with a text editor to make limited global changes to the contents of words, for example, to change riff to riff1 riff2 wherever it appears in part1, where these are word names:

```
"riff" "riff1 riff2" RENAME
"part1"GET
"riff1 riff2" "riff" RENAME
MAKE
```

You can use this to replace any word sequence by first replacing the sequence by a single dummy word of the same name, for example

```
"part1"GET
"C(EG)" []
MAKE
```

to make C(EG) a renamable word.

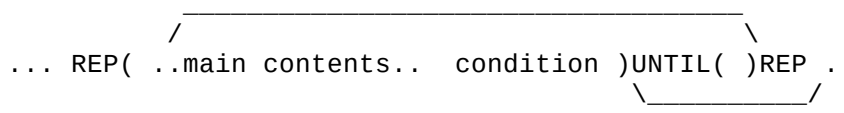
REP(start indefinite loop [] only

The instructions inside REP(...)REP are carried out an indefinite number of times.)UNTIL(can be included to leave the loop when a condition is met.

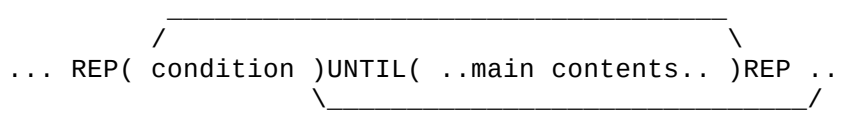
REP(...)REP is most commonly used to repeat something 'forever', that is, until the program is stopped by pressing ESCAPE.

'flag)UNTIL(' exits the loop (jumping to the first word after)REP) if the flag is ON. The flag can be the result of a condition test, just as with IF(.

The condition and)UNTIL(can be put anywhere in the loop: with them at the end, the loop acts like the 'REPEAT UNTIL' of other languages, doing the main contents at least once:



With them at the start, it acts like a 'WHILE REPEAT', not doing the contents at all if the condition is satisfied on entry:



You can combine these two by putting the)UNTIL(in the middle, and you can have more than one)UNTIL(in the same loop – they act entirely independently.

REP(...)REP can only be used inside words.

examples

```
"forever" [ REP( 0:CGd-E Fc/b )REP ]

"retwait" [ % wait for RETURN press
REP( #IN 13 #= )UNTIL( )REP ]

"ctrlhold" [ % utility command: CTRL holds music, SHIFT exits
REP(
 REP( -2 QKEY )UNTIL(           % exit if SHIFT pr essed
  -1 QKEY )UNTIL(               % exit if CTRL pressed
   IDLE )REP                    % until then, idle so players run
  -1 QKEY )UNTIL(               % exit main loop if SHIFT pressed
  ON PAUSE                      % else CTRL was pressed – pause
  REP( -2 QKEY NOT )UNTIL(      % wait until CTRL lifted
   IDLE )REP                    % until then, idle so players run
```

```
 OFF PAUSE                      % when CTRL lifted, release pause
)REP ]                          % return to top
...
RUN                             % start music playing
ctrlhold                        % use it
```

related words FOR( )UNTIL(

further information

Up to 23)UNTIL(s are allowed, depending on the number of structures within the loop.

)UNTIL(treats all non-zero values as ON, so you can directly test a number for non-equality to zero.

)REP end indefinite loop [] only

REP(...)REP encloses words that are to be executed an indefinite number of times, with)UNTIL(providing a conditional exit from the loop.

See REP(for more information.

$REV reverse the order of characters [] only

```
string $REV -> reversedstring
```

$REV reverses the order of the characters in the string, so the last (right-most) becomes the first (left-most) and so on. One use is to get access to characters at the right end for operations such as $CHR.

examples "hello" $REV leaves "olleh"

related words $12

RVOICES set voices range

```
startnumber endnumber RVOICES
```

RVOICES sets the voice range for the current ensemble, that is the range of voices that will be selected by a future EVERY VOICE command, and also itself selects all voices in the range. See VOICES for the operation of the voice range.

RVOICES is used, as is VOICES, to direct future sound settings such as instruments to all voices in a certain group at once.

whereas VOICES sets a range that starts with voice 1 and goes up to the number specified, RVOICES allows both the start and finish number to be specified, so is more useful where there are two voice sub-groups requiring different seettings. In fact,

number VOICES is equivalent to 1 number RVOICES

related words VOICES VOICE

example

```
3 5 RVOICES briteins  % send instrument to voices 3, 4 and 5
```

further information

The startnumber and endnumber may be any number from l to 12, the highest-numbered voice position available. If the startnumber is higher than the endnumber, then 0 VOICES, the empty voice range, will be set.

SAVE save program command string SAVE

SAVE saves all user words as a program file.

If the program is complete, it is usual for it to include a user word called RUN which runs the program.

example "piece2"SAVE

related words LOAD

further information

SAVE compacts free memory, stops players 1-10 and all sounds, and resets player 0's music action chain.

SAVE also saves any public editor data present.

SCORE prepare for music words

SCORE resets the players' music environment values, preparing it for music event words (notes, rests etc.). SCORE is used at the start of every section of score that uses music words, making sure that settings made in the previous section are cancelled.

The full effect of SCORE is:

```
   K( )K  0 BAR  1;  48,  0:  64=L  15'L  0@  SIMPLEACT
```

plus cancellation of any pending effect of +, =, –, ~, !, (, +L and -L, plus zeroing of the | word's ticks/bar count.

SCORE does not affect the tempo at all, neither resetting it or cancelling any +T or -T effect in progress.

example SCORE K(+F)K 192 BAR % signature

"sig" [SCORE K(+F)K 192 BAR]

related words SIMPLEACT K(BAR ; , : =L +L -L 'L @ ACT(

SHARE select voice ensemble

ensemblenumber SHARE

SHARE selects the ensemble to be used by this player, and carries out an EVERY VOICE to select all voices in the ensemble's current range. Though each player initially uses its own ensemble (group of voices), that is, the ensemble of the same number, SHARE can be used to select any of ensembles 0 to 10 for the player, causing sound commands (direct and from music events) to be sent to this ensemble along with those from any other players that have selected the same one.

SHARE is commonly used as a command to select a particular player's ensemble for adjusting its voices while a piece is playing, and as an instruction to set up players' voices from a single point in one 'master' player.

examples

```
   2 SHARE             % select ensemble 2 ('player 2' s voices')

   1 SHARE string      % change all voices of player 1 to 'string'

   3 SHARE 1 VOICE ins % change player 3's voice 1 only

   PNUM SHARE          % return to using this player's ensemble
```

related words P(PNUM VOICE

further information

Here are some of the advanced uses of SHARE:

1 Sharing one ensemble between two players which use it for

alternate passages, as an alternative to merging the scores

2 Using a single ensemble from more than one player simultaneously, for special effects or advanced score structures

3 Switching a player between alternative ensembles for instant instrument changes

When a player is created (by its first P(...)P), it uses its own ensemble, that is, effectively carries out a PNUM SHARE.

When using SHARE for complicated effects, you should remember that the VOICE setting (voice selection) is associated with the player (obviously), but the VOICES setting (voice range) is associated with the ensemble. You will normally only change an ensemble's VOICES setting from one player (usually the 'owning' one, the one of the same number), and in particular, not from a command entered while music is playing.

When arranging two or more players to use a single ensemble, usually one (often the 'owner') will be elected to set up the voices with instruments – take care that no other send sounds to the voices before this stage, making reference to the GO operation sequence if necessary.

SHOW show user words command

SHOW gives an alphabetical list of the names of the user words, followed by a count of them.

example

```
 %SHOW
RUN     act     all     next
riff2   riff    start   sync
wait
9 words
 %
```

related words RENAME DELETE

further information

It is possible for more than one word of the same name to exist in a single program. RENAME can create this situation. Commands will access the most recently-created version, and this can be renamed to allow access to the others.

The user word list is searched in reverse order so that preference is given to a longer name rather than a leading substring of it. This ensures that with the words 'fred' and 'freda' on the SHOW list, the input 'freda' would be interpreted as the word 'freda' not the Words 'fred' and 'a'.

SIGN test number is negative

number SIGN -> flag

SIGN takes a number and leaves flag indicating its sign – ON if negative, OFF if positive or zero.

examples ... SIGN IF(...)IF % do if negative

related words #> #<

further information

SIGN is a faster alternative to 0 #< .

SIMPLEACT remove all music actions

SIMPLEACT removes all music actions on this player. It is used to initialise the player's music action chain to the initial, empty, state.

Other words that remove music action include SCORE, which performs SIMPLEACT and initialises all other music environment values, and ACT(, which allows an action to be removed individually by reference to their definitions.

related words SCORE ACT(

further information

SIMPLEACT is useful for clearing all actions without having to know which are in use and then clearing them individually.

SP print a space

SP prints a space on the screen. It is a shorter and clearer equivalent to 32 #OUT. SP is commonly used to separate numbers on one line.

example 5 NOUT SP -4 NOUT prints 5 -4

related words #OUT NL

STOP stop program

STOP stops the program (including players 1-10) and all sounds.

STOP is used either as a command to stop a piece that is playing, or as an instruction in a word definition to end execution immediately and return control to the % prompt.

When a piece ends naturally, the memory used by players 1-10 is not freed (since they could be waiting for further instructions), but STOP frees this memory, making sure is it available for other uses.

examples

```
%RUN                        % command to start playing
%STOP                       % end piece prematurely

... #IN 13 #= IF( STOP )IF  % end on RETURN
```

further information

STOP does the following:

* silences all sounds
* stops players 1-10, and frees the memory used by them
* sets OFF PAUSE and OFF HALT
* sets player 0's note context to normal
* sets player 0 to use its own voices (ensemble no. 0)

It does not free voices.

$STR convert number to decimal string represenation [] only

```
number $STR -> string
```

$STR converts a number to the string of characters representing the number in decimal, including a leading minus sign if the number is negative. It is commonly used to format or process the text of a number before printing it, for example to put it in a fixed size field with $PAD.

examples

```
-425 $STR   produces   "-425"

"fnout" [ % number fieldnumber fnout
% print number in field of specified width
```

```
   #12 $STR    % convert to number
   $PAD        %pad to specified number of spaces
   $OUT ]      % print string
```

related words &$STR VAL $PAD

&$STR convert number to hex string representation [] only

number &$STR -> string

&$STR converts a number to the string of characters representing the number in hexadecimal (not including the & sign). It is commonly used to format or process the text of a number before printing it, for example to put it in a fixed size field with $PAD.

examples

```
   254 &$STR produces "FE"

   "&fnout" [ % number fieldnumber fnout
   % print number, with & ,in field of specified width
   #12 &$STR   % convert to number
   "&" $+      % add & to start
   $PAD        % pad to specified length with spaces
   $OUT ]      % print string
```

related words $STR &VAL $PAD

$STRIP remove leading spaces from string [] only

string1 $STRIP -> string2

$STRIP removes any spaces that are on the start (left end) of the string.

$STRIP is often used in the processing of string input, in particular to remove leading spaces from a number in string form, before converting to numeric form by VAL or &VAL (these do not strip leading spaces themselves) .

examples

```
   "   hello" $STRIP    produces   "hello"

   " 10"      $STRIP    produces   "10"

   " 10"      VAL       produces   " 10" OFFflag % no number found
   " 10" $STRIP VAL     produces   "" 10 ONflag  % number found
```

related words $PAD VAL &VAL

=T set tempo

number =T

=T sets the tempo in beats per minute. The tempo is the rate at which musical time passes, that is, a measure of how long a single timebase tick (one unit of DURATION or ',') lasts. It is global, that is, applies throughout the system, to all players.

The 'beat' is the current ',' setting, so an =T instruction is often preceded by a ',' setting to make it complete. In fact, the final tempo, in ticks per minute, is the beat value (ticks per beat) mutiplied by the =T number (beats per minute), so either may be used to control the tempo. One example of this is a sequence of music events starting with =T – the resulting speed is independant of the prevailing beat setting:

```
number,  125=T XXXX         % speed independent of number
```

=T cancels any +T or -T changes in progress.

READY sets the tempo to '48, 125 =T', which is exactly 100 ticks per second, or 6,000 ticks per minute.

examples

```
READY 100 =T       % sets tempo to 100 crotchets per minute

1, 50000 =T        % set tempo to 50000 ticks per minute

"tempo" [ 60 =T ]  % will always set the tempo to 60 beats
                   % per minute, regardless of ',' value.
```

related words , +T -T READY

further information

The range of =T control is 92 to 65535 ticks, or with a beat setting of '48,' , 2 to 1365 crotchets per minute.

=T issues a tempo sound event which is interpreted by the current time server. Non-standard time servers may provide alternative functions for =T.

+T increase tempo changenumber
beatsnumber +T

+T increases the tempo by the specified amount, over the specified number of beats (',' units). The change number is exponential in effect, with 0 giving no change, 64 doubling the tempo and -64 halving it. The range of the change number is -127 to 127.
+T is used for a range of relative tempo changes, both gradual ('accelerando' and 'rallentando') or instantaneous.

The exponential scale ensures that tempo changes of opposite sign are entirely complimentary - any change can be reversed with a change of the same magnitude and opposite sign. Also, tempo changes can be specified as fractions, using one change for the top number, followed immediately by a second for the bottom.

Each +T instruction cancels any +T or -T effect in progress, and begins its effect from the tempo reached at that time.

examples

```
64 0 +T      % instantly double tempo

20 16 +T     % accelerando (gradual speed-up)
C//D E/c/    % of 20 units over 16 beats
EFG/ A///

-37 8 +T     % decrease by 50% over 8 beats ('rallentando')
XXXX XXXX
XXXX XXXX
37 0 +T      % instantly restore tempo ('a tempo')
```

related words , =T -T

further information

The new tempo, expressed as a percentage of the old, is given by:

$$100 \times 2^{(\text{changenumber} / 64)}$$

Here is the new tempo for each of a range of change values,

expressed as a percentage of the old:

changenumber	relative tempo	changenumber	relative tempo
-64	50	0	100
-56	54.5	8	109
-48	59.5	16	119
-40	64.8	24	130
-32	70.7	32	141
-24	77.1	40	154
-16	84.1	48	168
-8	91.7	56	183
		64	200

A decrease can be achieved either by +T with a negative value, or -T with positive value.

The precise percentage tempo change is:

$$100 \times 2^{(\text{changenumber} / 64)}$$

The exponential scale allows tempo changes to be combined for greater or more-convenient control. In addition to changes that return to the original after a time, the applications include:

* multiple precision – a second immediate tempo change adds extra precision for wide changes
* fractional changes ('metrical modulations') – one change does the top number, and a second does the bottom, for example:

```
                % total tempo change of 3/2
101 0 +T        % x 3
-64 0 +T        % / 2
```

+T and -T issue tempo sound events which are interpreted by the current time server. Non-standard time servers may provide alternative functions for +T and -T.

-T decrease tempo

changenumber beatsnumber -T

-T decreases the tempo by the specified amount, over the specified number of beats (',' units). The change number is exponential in effect, with 0 giving no change, 64 halving the tempo, and -64 doubling it.

-T is used for a range of relative tempo changes, both gradual ('rallentando' and 'accelerando') or instantaneous. Its effect is

identical but opposite to that of +T. See +T for further information.

related words =T +T

TYPE type word definition on the screen command

namestring TYPE

TYPE types the contents of the word on the screen, as it would appear in a text editor. It is useful for quickly examining words without using the editor, which may have other text in it.

TYPE also works on editable module words, for example, preset instruments.

example "sig" TYPE

UNTIL(exit from indefinite loop [] only

REP(...)REP encloses words that are to be executed an indefinite number of times, with)UNTIL(providing a conditional exit.

See REP(for more information.

UNUSED make voice(s) unused

UNUSED makes the currently selected voice or voices unused, freeing a voice of that type for use elsewhere. UNUSED voices make no sound, and do not respond to sound commands or music events.

UNUSED is used to recover voices when they have been finished with.

example

```
3 VOICES ins1       % three voices
...
3 VOICE UNUSED      % free 3 for use elsewhere

3 VOICES ins1       % three voices
...
ON VOICE UNUSED     % free all voices on this player
2 VOICES ins2       % assign two for smaller chords
```

related words VOICE VOICES RVOICES

further information

Since, assigning any voice will replace the voice previously in use, UNUSED can be thought of as a voice type itself, but one which has no limit on the number in use.

VAL convert string to unsigned decimal number [] only

```
string VAL -> remainingstring number ON    if found
           -> remainingstring OFF          if not found
```

VAL converts the string representation of a decimal number into a number on the stack, leaving the remainder of the string. The number is left with ON on top, or if no number was found, OFF is left. VAL is used with $IN to let the program accept numbers from the keyboard while running.

VAL expects the first character of the string to be a decimal digit: it does not ignore leading spaces or recognise a minus sign. Leading spaces should be removed beforehand with $STRIP.

examples

```
"10 20" VAL     produces  " 20" 10 ON

"nin" [ % input decimal number
        % nin -> number ON  if legal number found
```

```
        % nin -> OFF        if legal number not found
   $IN     % get string
   $STRIP  % remove leading spaces
   VAL     % convert to number
   $2 ]    % discard remaining string

   "lnin" [ % get a line of numbers entered by the user
            % lnin -> number-n ... number2 number1 countnumber
   $IN           % get line
   0             % initial count of numbers found
   REP(
    $STRIP VAL   % look for number
    NOT )UNTIL(  % exit if none found (non-valid, or end of line)
    #12          % swap foud number and count, so count is on top
    1 #+         % increment count
   )REP
   $2 ]          % drop remainder of string
   "trylnin" [ % test routine
   REP(
    lnin               % get list of numbers, with count on top
    FOR( NOUT SP )FOR  % print them (in reverse of input order)
    NL )REP ]          % repeat for ever
```

related words $STR &VAL

&VAL convert to unsigned hex number [] only

```
   string &VAL -> remainingstring number ON   if found
               -> remainingstring OFF         if not found
```

&VAL converts the string representation of a hexadecimal number (with no 5) into a number on the stack, leaving the remainder of the string. The number is left with ON on top, or if no number was found, OFF is left. &VAL is used with $IN to let the program accept numbers from the keyboard while running.

&VAL expects the first character of the string to be a hexadecimal digit: it does not ignore leading spaces or recognise a minus sign. Leading spaces should be removed beforehand with $STRIP.

examples

```
   "OF 3C" &VAL   produces  " 3C" 15 ON

   "&nin" [ % input hex number
            % &nin -> number ON  if legal number found
            % &nin -> OFF        if legal number not found
   $IN $STRIP $VAL $2 ]

   "dhnin" [ % get number input in either form: dddd or &hhhh
```

```
          % dhnin -> number ON  if legal number found
          % dhnin -> OFF        if legal number not found
   $IN $STRIP          % get line and strip spaces
   1 $- ASC            % get initial character as ascii number
   #11 "&" ASC #=      % if it is & ...
   IF( &VAL )ELSE(     % treat as hexadecimal
    $CHR $+            % else out the initial characetr back on,
   VAL )IF             % and treat as decimal
   $2 ]                % discard remainder of string
```

related words &$STR &VAL

VOICE select voice(s)

voicenumber VOICE

VOICE selects the specified voices to receive sound instructions issued by the player. The voices are numbered from 1 upwards. VOICE is used to send instruments and individual sound instructions to a particular voice.

VOICE should not be confused with VOICES, which selects more than one voice simultaneously.

Note that VOICE selects the voice for sound instructions, not music events (notes, hits, rests and ties). Music event voice selection is done using ';'.

The voice selection is only valid for the sound instructions up to the music event, since these instructions set it back to EVERY VOICE. You must use VOICE again before the next group of sound instructions, for example

```
..music..  n VOICE ..sound.. ..music..  n VOICE ..sound..
```

EVERY VOICE has the special effect of selecting voices in the current voice range (set by VOICES or RVOICES) simultaneously, so that the following sound instructions are sent to all of them.

examples

```
... sect1  ON VOICE instru  sect2 ...      % new instrument

... sectl  1 VOICE instr1 2 VOICE instr2  sect2 ...
             % different instruments on each of 2 voices

3 VOICES tom-tom    % 3 voices with the same instrument...
 1 VOICE -7 PITCH   % but different variations
 2 VOICE  0 PITCH
 3 VOICE 12 PITCH
```

related words VOICES RVOICES

further information

The following argument values are allowed:

```
 OFF    select no voices
 1-12   select numbered voice
 EVERY  select all voices in current range (set by VOICES/RVOICES)
```

Each music event achieves its effect by sending sound instructions which are normally invisible to the user, and it uses UOICE to direct them to the voice specified by ';'. The event finishes by executing EVERY VOICE, thereby leaving the voice selection in a defined state. Because of this, any sound instruction without a VOICE instruction that lies after a music event will be applied to all voices in the current range.

Note that SHARE and VOICE can be used together to select any voice – in effect, SHARE sets the top digit (the ensemble number) and VOICE the bottom digit of a complete two-digit voice number.

Hore care is needed in using VOICE when DURATION (or '\') is in use to move up and down musical time. Under these conditions, the voice selection cannot be relied upon after a DURATION because a conflicting voice selection could be left in this time interval by a VOICE from later or earlier in the program. The rigorous method is to confirm the voice selection with VOICE between each DURATION and the following sound instruction.

VOICE! set voice settings in frame

```
voicenumber VOICE!   or   EVERY VOICE!
```

VOICE! sets all non-zero voice values in the current music action frame to the specified value. In fact, it will operate on any seven numbers below the current frame pointer, but its use is usually confined to action frames.

VOICE! is used to set the destination voice prior to calling ACT, usually from within ACT(...) ACT. The standard music action frame has three voice values (see ACT for details) each of which is either OFF, or the the voice number, depending on the event type (note, rest etc.). VOICE! changes the voice number only, leaving OFF values unchanged and so preserving the event type.

examples

```
   ... 2 VOICE! ...  % set voice to 2
```

```
"vtop" [ % flag vtop
         % redirect each voice to player of the same number
         % i.e. voice 1 to player 1, voice 2 to player 2 etc.
         % used for passage of large chords, for example
20 AND ACT(
5 FVAR #?         % get event voice (actually, gate voice)
SHARE             % select player
1 VOICE!          % change all voice numbers to 1
ACT               % execute event
)ACT ]
```

related words VOICE ACT(MVAL

VOICES set number of voices

```
number VOICES
```

VOICES sets the 'range' of voices to include those from 1 to the number given, so that these voices will be selected together by a future EVERY VOICE command, and also itself selects all voices in this range. Whenever an EVERY VOICE is used, all voices in the range set by the previous VOICES will be selected. The range remains set until the next VOICES, RVOICES or READY.

VOICES is used to direct future sound settings such as instruments to all voices in a group at once, in particular, to the number of voices specified, starting with voice 1. It should not be confused with VOICE which selects voices, but doesn't set the voice range.

The voice range belongs to the ensemble (the object selected by SHARE) not the player itself, so VOICES (and RVOICES) is directed to the ensemble currently selected by the player. This means that when two players are accessing the same ensemble (after one has SHAREd the other), they share a common voice range, and one player could inadvertently change the effect of the other's sound instructions by altering the voice range. For this reason, when accessing another player's voices through SHARE, you should not normally use VOICES (or RVOICES).

examples

```
READY 8 VOICES ins1

...
4 VOICES ins1       % four identical voices,
3 VOICES -12 PITCH  % but 1 to 3 have pitch set
4 VOICES            % restore range for future EVERY VOICEs
...
```

related words VOICES VOICE RVOICES

further information

The VOICES number should be in the range 0 – 12.

READY sets the voice range of each ensemble to 0 VOICES.

WIND advance time

ticksnumber WIND

WIND instantly advances the system’s real time by the number of timebase units given. Its uses include 'winding on' a piece of music by a certain amount, and advancing time manually when the timebase is stopped by HALT.

Note that it is not often useful to use WIND in the ON PAUSE state, since sound execution is completely halted.

examples

```
300 WIND      % wind on by 300 ticks

"bw" [ % barsnumber bw   % wind on by n 4/4 bars
48 #*         % ticks per beat
4 #*          % beats per bar
WIND ]

"part9" [ % simple 'timebase distortion' adding global 'swing'
          % feel by compressing second half of beat
SCORE 24, /   % move to middle of crotchet beat
REP( 12 WIND  % leap through next 12 ticks (vary to taste)
48,/ )REP ]   % wait for one crotchet to pass
...
"part1" [ % demonstration
SCORE
4 FOR( 48, XXXX )FOR       % crotchet hits unaffected
REP( 24, XX XX XX XX )REP  % quavers are played as triplets,
                           % for example, 12, X/X/ X/X/
                           % is played as: 12, X/X X/X
```

WRITE display text of all words command

WRITE writes the text of all user words to the screen, from where it can be printed, or sent to a file with *SPOOL. The spooled text can be converted back to a program using the *EXEC command.

WRITE adds an empty definition for each word before writing the real definitions in alphabetical order. This ensures when recreating the program from the text, that a word definition can use another word (or itself) which may not have been completely compiled at that point.

examples

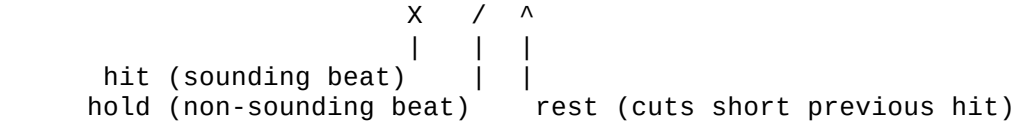
```
CTRL-B WRITE
CTRL-C        % print program

*SPOOL text         % spool program text to
WRITE               % file 'text' for editing
*SPOOL              %   with word processor
...
AMPLE               % recreate program after
*EXEC text          %   editing text
```

related words TYPE

X play hit

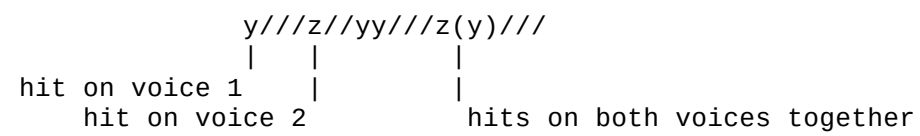
X plays a 'hit' on the current music voice. X is very like the note letters A-G except that it does not set the pitch. It is used to play hits ('beats') in percussion scores and to restrike the last note in pitched music scores.

The hit takes as its length the basic length (set with ','), though you can extend it with the hold symbol, /. Many percussion instruments have a sound that dies away immediately, so this will only affect the amount of time to the next hit, not the length of the sound. Holds are used, rather than rests, to mark silent beats between hits, allowing the sound to die away naturally. A rest cuts a long sound short.



Hits can be included in chords exactly like notes, for multi-voice percussion scores. Alternatively, a user 'hit' word can be defined for each voice – "y" [n;X] where n is the voice number.

Chord brackets are then only used when more than one hit plays on the same beat:



The pitch of a percussion instrument is determined as part of its definition, with PITCH.

Used in normal (pitched) scores, X serves to repeat the last note on that voice. It is useful for repeating chords using ON; .

examples
```
XXXXX//XX^//X/X/   % rhythm with one short hit
y///z//yy///z(y)///        % two-voice rhythm
                           % with one double hit
C(GE) EVERY;XXX 1; % repeated chord
```

related words ;

further information

The length of the hit is added to the bar's total of note lengths for checking by the next bar line.

X calls the player's current music action list, passing the following stack frame:

description	value	default destination
pitch voice	OFF	VOICE
pitch	undefined	PITCH
level voice	event voice	VOICE
level	calculated level	VEL
gate voice	event voice	VOICE
gate	ON	GATE
duration	',' setting	DURATION

On return from the action list, it executes EVERY VOICE to return the voice selection to a defined state.

If ~ is applied to X, it reduces it to a hold (with the default music action) – see ~ for details.

XOR exclusive-OR bits of numbers

```
number1 number2 XOR -> number3
```

XOR performs the logical exclusive-OR operation on the bit patterns of the two numbers. Each bit in the result is only 1 if the corresponding bits in the input numbers are different.

XOR is used both as a bit-wise operator for manipulating bit patterns, and as a logical operator for flags:

bit1	bit2	bit3	flag1	flag2	flag3
0	0	0	OFF	OFF	OFF
0	1	1	OFF	ON	ON
1	0	1	ON	OFF	ON
1	1	0	ON	ON	OFF

examples

```
&1234 &FF XOR   produces   &12CB
in binary:
          0001001000110100 XOR
          0000000011111111
produces  0001001011001011

... &FFFF XOR ...    % invert bits in number (bit-wiseNOT)

... 3 XOR ...        % toggle 1 -> 2, and 2 -> 1
```

related words AND OR NOT

Index

!	91
$+	49, 99
$-	49, 99
$12	100
$2	100
$CHR	49,133
$IN	53,55,150
$OUT	55,169
$PAD	49,170
$REV	49,183
$STB	188
$STRIP	49,52,189
&	101
&$STR	189
&NOUT	55,167
&VAL	195
'L	156
)ACT	127
)ELSE(	143
)FOR	145
)IF	152
)K	155
)P	172
)REP	183
)UNTIL(	193
+L	157
+T	191
-L	158
-T	192
<cr>	90
<space>	90
=L	156
=T	190