ACORN COMPUTER

Music 500 SYNTHESISER USER GUIDE

The Music 500

User Guide

Part no 0450,000
Issue no 1
Date November 1984

WARNING: THE MUSIC 500 MUST BE EARTHED

Important: The wires in the mains lead to the synthesiser are coloured in accordance with the following code:

Green and yellow	**Earth**
Blue	**Neutral**
Brown	**Live**

As the colours of the wires may not correspond with the coloured markings identifying the terminals in your plug, proceed as follows:

The wire which is coloured green and yellow must be connected to the terminal in the plug which is marked by the letter E, or by the safety earth symbol or coloured green, or green and yellow.

The wire which is coloured blue must be connected to the terminal which is marked with the letter N, or coloured black.

The wire which is coloured brown must be connected to the terminal which is marked with the letter L, or coloured red.

If the socket outlet available is not suitable for the plug supplied, the plug should be cut off and the appropriate plug fitted and wired as previously noted. The moulded plug which was cut off should be disposed of as it could be a potential shock hazard if it were to be plugged in with the cut off end of the mains cord exposed. The moulded plug must be used with the fuse and fuse carrier firmly in place. The fuse carrier is of the same basic colour* as the coloured insert in the base of the plug. Different manufacturers' plugs and fuse carriers are not interchangeable. In the event of loss of the fuse carrier, the moulded plug MUST NOT be used. Either replace the moulded plug with another conventional plug wired as previously described, or obtain a replacement fuse carrier from an authorised BBC Microcomputer dealer. In the event of the fuse blowing it should be replaced, after clearing any faults, with a 3 amp fuse that is ASTA approved to BS1362.

*Not necessarily the same shade of that colour.

Exposure
Like all electronic equipment, the Music 500 synthesiser should not be exposed to direct sunlight or moisture for long periods.

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

First published 1984
Published by Acorn Computers Limited

Contents List

1 Introduction

The **Music 500** is an exciting breakthrough in computer-based music synthesisers. It offers you a vast range of exciting sounds and facilities, without you needing to be either a computer programmer, or a music scholar.

The Music 500 makes it easy by providing a novel way of creating different sounds and notes. All the standard devices necessary in music composition are made as easy as possible to use, whether you are typing directly at the keyboard, or composing something for playing later.

Here are some of the facilities you have at your disposal:

- Seven position stereo output
- Up to sixteen notes can be played at once
- Thirteen programmable preset waveforms
- Thirteen programmable envelopes, plus any you build yourself
- special effects including ring modulation, frequency modulation, synchronisation and phasing
- pitch variation, either large or infinitesmally small
- individual volume control of each component of each instrument
- up to 8 independent performers can play at once, plus keyboard commands

The Music 500 is controlled by words letters and numbers typed in at the keyboard which constitute the 500's own language called AMPLE. With even a small vocabulary, and a smattering of grammar, you can make use of many of the 500's facilities.

The Music 500 uses the BBC Microcomputer to control it, and to present AMPLE to you the user. A good quality amplifier and speakers connected to the 500's audio output completes the system. Enjoy yourself!

About this manual

The *Music 500 User Guide* takes you through the system in three stages, the first gets you started, the second is a Tutorial, and the third stage explains the more advanced features of the Music 500. The reference section at the back contains a dictionary of AMPLE words.

Conventions used in this manual are as follows:

Ordinary text appears like this.
Emphasised text appears *like this*.
Words etc. typed at the keyboard, AMPLE words and anything which appears on the screen appears **like this.**
Words like **RETURN** refer to the key on the keyboard, as opposed to the letters R E T U R N.

Screen display

The normal screen display you will use is the Teletext display, or MODE 7, on the BBC Microcomputer. In this display mode, the keyboard characters [ ] ^ appear as ← → ↑ on the screen, and thus appear as ← → ↑ in this manual.

2 Installation

Checklist of items

The Music 500 box contains the following items, including this book:

- The Music 500 synthesiser unit, with a captive mains lead.
- A cassette, containing the AMPLE language and some demonstration music.
- A guarantee registration card.

Other equipment you will need

- A BBC Microcomputer model B
- A monochrome monitor.
- A suitable cassette recorder, and connection to the BBC Microcomputer.
- A good quality stereo hi-fi amplifier, or stereo monitor amplifier.
- A good quality screened audio lead to connect the Music 500 to the amplifier.
(ask your dealer to provide you with this lead, which should have a five pin DIN plug at one end, and a suitable connection for your amplifier at the other.)
- A pair of speakers or headphones to plug into the amplifier.

Choosing loudspeakers

Your loudspeakers may be subjected to some heavy treatment. Make sure that your speakers can handle the full power of the amplifier, or else keep the volume low.

Checking the computer

Before connecting the Music 500 to your BBC Microcomputer, check the following conditions on your BBC Microcomputer:

- **Operating System** This should be version 1.2. Type

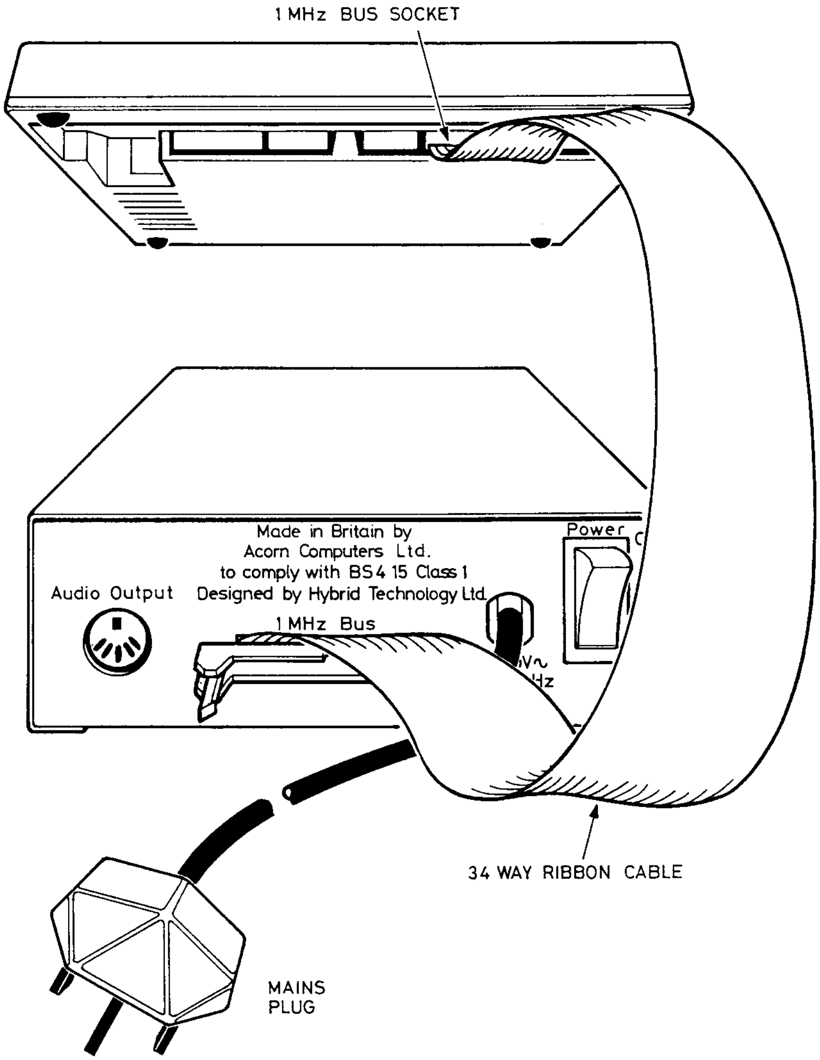
***HELP RETURN**

To display the version number.

- **Second Processors** Any Second Processor must be switched off..

Connecting the Music 500

With everything turned off, plug the flat cable from the Music 500 into the socket marked '1MHz Bus' on the underside of the computer. This is illustrated below. Make sure that the arrow mark on the cable connector aligns with the arrow on the '1MHz' Bus connector.

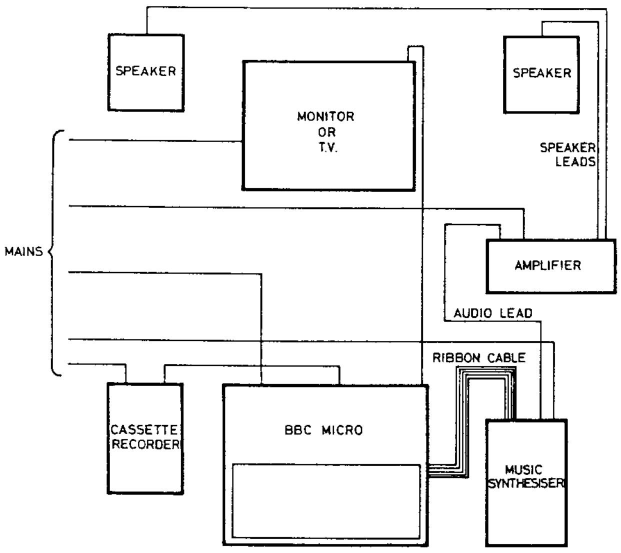

1 MHz BUS SOCKET
Made in Britain by
Acorn Computers Ltd.
to comply with BS4 15 Class 1
Designed by Hybrid Technology Ltd.
Audio Output
1 MHz Bus
Power
34 WAY RIBBON CABLE
MAINS
PLUG

Connect your cassette player to the computer via the socket marked 'Cassette', and connect a TV or monitor to the relevant video output on the computer.

Connect the amplifier audio lead between the Music 500 five pin DIN socket and the input to your stereo amplifier. The input you choose on your amplifier should be a 'line level' input such as 'AUX', 'TUNER', 'TAPE PLAYBACK' - *not* a sensitive input like 'GRAM', 'PHONO', 'MIC' etc.

Connect the speakers to the amplifier, (or headphones if you prefer).

The complete system should look something like this:



Avoiding hum and noise

The Music 500 produces a very high quality audio output which is remarkably free from noise. If you notice an appreciable noise level which cannot be attributed to your amplifier alone, then return to this section after testing.

As with any audio installation, the way in which the earthing is organised is very important. Bad earthing can lead to 'hum' problems, and to the presence of background noise. Although audio earthing is generally considered a bit of a black art, here are some tips if you experience either of these unwanted effects after testing the system.

Try to plug the computer, the Music 500 and the amplifier into mains outlets as close together as possible (ideally, use a four way mains adapter block).

If you are using a tuner/amplifier, don't situate it very close to the computer, which may cause slight interference.

Switching on and testing

With the amplifier volume set to zero, switch everything on.
If the computer doesn't start up in BASIC, type

***BASIC RETURN**

to get into BASIC. If the filing system in use is other than cassette, then type

***TAPE RETURN**

Now insert the Music 500 cassette *the 'A' side uppermost* into the cassette player, and type

CHAIN "AMPLE" RETURN

Press the PLAY button on the cassette, and wait while the Music 500's language (AMPLE) loads. When this is complete, the program will run, and the following message will appear on the screen:

```
AMPLE Model BCE Version 1.0
(C) 1984 Hybrid Technology
 %_
```

The **%** sign indicates that the Music 500 is ready to receive your instructions, and you can now stop the tape.

Disc transfer

At the very beginning of the cassette is a short program called "TAPEDISC". This program is to enable you to copy the complete contents of the tape onto disc. To do so, insert a formatted blank disc into your disc drive, and from BASIC type

CHAIN "TAPEDISC" RETURN

and press the PLAY button. When the program has loaded, it will start copying straight away.

Testing

Type in the following to generate a continuous tone:

SOUND SCORE C RETURN

Increase the amplifier volume until you hear a tone through both loudspeakers.
If you hear nothing, check that:

- the amplifier is switched to the input you are using
- the flat cable to the computer has been plugged in the right way round
- the lead from the Music 500 to the amplifier is the right type.

Now press **ESCAPE** to silence the Music 500, and turn to the next chapter.

3 Demonstration pieces

As you have seen, side 'A' of the cassette contains the AMPLE language. Side 'B' contains a number of pieces of music already written for the Music 500, and apart from their entertainment value, they should serve to show some of the capabilities of the machine.

First, turn the cassette over to side 'B' and rewind the tape back to the beginning. Type

"index" LOAD RUN RETURN

This displays a list of the names of the demonstration pieces on side 'B'.

To listen to any of these, type the name of the piece, (note there are no capital letters) in quotes, followed by **LOAD RUN**, press **RETURN** and start the tape. For example

"popcorn" LOAD RUN RETURN

will play the piece entitled 'popcorn' on the cassette.

4 Getting started

Introduction

In this chapter, we will start using the Music 500 by trying out some sounds and some simple tunes, and learning a few basic rules. Don't expect too much in-depth explanation here, as this is the job of the Tutorial section later. If you don't understand any of the musical terms you find in the rest of the manual, you'll find a Glossary at the back.

Note: From now on, you must remember to type **RETURN** at the end of lines entered.

Please insert the cassette into your cassette player on side 'A', and type

"preset1" LOAD RUN

and start the cassette near the end of the AMPLE file on the tape. This gives the Music 500 some predefined envelopes and waveforms, which you will need shortly.

******** typing **SOUND RETURN** will mostly restore the Music 500 to its condition on startup, do this whenever you are totally confused, or after listening to oone of the demonstration pieces BEFORE continuing. If this does not work, type **AMPLE RETURN** which will completely reset everything to the startup state. However, if you do this, you will need to reload the "preset1" file again.

Playing some notes - I

Part II of this section can be found in the next chapter.

Going up and down

Type

SOUND SCORE

This gets the 500 ready to play some simple music. Now type

CDEFGABC

You will hear a sequence of notes play one after the other, going up in pitch. This is the scale of C major, starting at middle C and ending at the C one octave above. If you are unsure about any of the words in this sentence, turn to the Glossary which shows part of a piano keyboard, and will be of help.

Notice that the last note is still sounding. Type

↑

and the note is silenced. The **↑** symbol stands for a 'rest' which is a silent note (*not* a break in the music). Now type

bagfedc↑

This time the notes go down the scale, and because you typed a rest at the end, the music stops sounding. Putting all this together in one line:

cDEFGABCbagfedc↑

this is what an up-and-down scale in C major looks like. So, to play a note higher in pitch to the last one, enter upper case letters, and to go down in pitch enter lower case letters. Try this:

CDbCD EFedc DcbC↑

This plays the first part of 'God Save the Queen', with all notes the same length, and with no repeated notes.
Note the spaces - these have no effect at all, and so can be used to 'section off' parts of the tune on the screen.

Fixing the pitch

To make sure that a note plays at a particular pitch which isn't dependant on the last note, the **:** word fixes the pitch. It is used with a number before it, and the number determines in which octave the following note is to play. The middle octave is **0**, negative numbers are lower octaves, and positive numbers are higher octaves. For example:

0:C	is middle C
0:c	is the C below middle C
1:G	is the G above middle C
-2:e	a low E

Tunes should normally start with an octave setting, otherwise the previous note played may affect in which octave your tune starts.

The one exception to the upper/lower case rule is repeated notes: if a note is repeated with the same case, it always plays the same pitch. To improve on our national anthem:

```
O:CCDbCD EEFedc DcbC↑
```

Varying the note length

The length of notes is set by the , word. This word requires a number in front of it, and this determines the length. When you first start playing notes after **SCORE**, the length is automatically set to 48. Now let's try to imrove our national anthem:

0: 48,CCD 72,b 24,C 48,D EEF 72,e 24,d 48,c DcbC↑

Remember that the spaces shown have absolutely no effect.

Tying notes

An alternative to specifying new note lengths all the time is to 'tie' short notes together to make long ones. The tie word is /, and has the following effect:

24,C/ ↑ will play a C with the same length as **48,C ↑**

24,C// ↑ will play a C with the same length as **72,C ↑**

Now we can simplify our example:

0: 48,CCD 24,b//,C 48,D EEF 24,e//d 48,c DcbC↑

Sharps and flats

To play a sharp note, precede it with a **+**, and to play a flat note, precede it with a **-**. So **+C** is C sharp, and **-D** is D flat (both the same note!). Here are two ways of playing all the semitones in an octave:

0:C+CD+DEF+FG+GA+ABC↑

0:C-DD-EEF-GG-AA-BBC↑

NB: Sharpening and flattening a note like this ONLY affects that note, so

Cg-ec Cg-ec↑

is not the same as

Cg-ec Cgec↑

Sharpening and flattening all occurences of a note is dealt with in the Advanced AMPLE section under *Key signature*.

Changing the sound

Starting up

So far we have used one very simple sound for playing notes. The power of the Music 500 lies in the wide range of sounds it can produce, and these can be created and controlled with *sound words*. To start off creating some new sounds, and to provide you with a continuous tone to work with, type

SOUND ON GATE

and you will hear the tone. The **SOUND** word sets the sound to a basic starting state, and **ON GATE** turns the sound on - similarly, **OFF GATE** turns it off.
To silence the sound quickly, you can press the **ESCAPE** key.

Changing the pitch

With the sound on, enter

100 PITCH

Alternatively, you can continuously range up and down the **PITCH** numbers by using a special command - **SCAN**. Type

SCAN PITCH

The pitch resets itself to middle C (**PITCH** no. **0**), and the number is displayed on the line below. To scan up and down in increments of one unit, use the 'colon' and 'close square bracket' keys. To scan in increments of 10 units, use the '@' and 'open square bracket' keys. To finish scanning, and leave the pitch at the value displayed (and heard), press **RETURN**.

Changing the tone

Changing the tone is done by using the **WAVE** word. The tone is governed by the waveform of the sound, and there are fourteen ready-made **WAVE**s, and the one you can hear at the moment is number **0**. To change the waveform, type

3 WAVE

The sound changes tone to **WAVE** number three. **SCAN** can be used again for speed, so type

SCAN WAVE

The original **WAVE**, number **0** is selected, but using the 'colon' and 'close square bracket' keys shows you what the others sound like. Again, press **RETURN** to leave scanning. For best effects, keep the pitch number at around -200 while you scan through the **WAVE**s.

Stereo positioning

The stereo position is currently in the middle, but it can be shifted to either side by the **POS** word. The number entered before **POS** determines the position, from **-3** (extreme left) to **0** (middle) to **3** (extreme right. The **POS** word can also be scanned with **SCAN**. Try typing

-3 POS

2 POS

etc.

Volume

To alter the volume of the sound, use **AMP**. This takes numbers from 0 to 128 (loudest), although the useful range starts at about **100** upwards. Thus

90 AMP

reduces the volume, and

128 AMP

restores the volume to the normal maximum level. **SCAN** works with **AMP** as well.

Playing tunes with the new sound

Now that you've created a more complex sound, type

SCORE

and enter a sequence of notes to try out the new sound. You can make any changes you like to either channel by typing them in.

Summary

The ESCAPE key

Pressing **ESCAPE** will immediately silence the Music 500. However, it is better practice to end a sequence of notes with a ↑.

SOUND and SCORE

Entering **SOUND** restores the sound type. **SCORE** resets the note lengths to the standard 'startup' values. To think of **SOUND** and **SCORE** as being two independent 'states' is mistaken. Once both commands have been entered, ie

SOUND ON GATE

to generate a tone to monitor the building of a new sound, and

SCORE

to enable you to play notes, you can freely interchange between changing the sound and entering notes.

How the commands are obeyed

You will have noticed by now that the Music 500 obeys an instruction by adding its effect to what has gone before. For example

```
1 CHAN -100 PITCH
1 CHAN 5 WAVE
1 CHAN 100 AMP
```

is the same as

```
1 CHAN -100 PITCH
5 WAVE
100 AMP
```

is the same as

```
1 CHAN -100 PITCH 5 WAVE 100 AMP
```

because having specified channel number 1, all following instructions will automatically be applied to channel 1 until you specify a different channel number. However, you will have to specify the channel again after playing notes.

The order within an instruction

You will also have noticed that you enter the number first, and the instruction the number refers to second. This is true for all AMPLE instructions, whether the particular value is a number or a word.

All commands are always entered in upper case, so **CHAN** cannot be netered as **Chan** or **chan**.

Spaces

Mostly, spaces can be inserted into what you type, or left out. Their main use is to make the displayed text readable. The only exception to this is where two AMPLE words without a separating space may make a third word, for example

```
ALL CHAN SCAN PITCH
```

will allow you to change the pitch on all the channels currently in use, but

```
ALL CHANSCAN PITCH
```

will give an error message, because **CHANS** is also an AMPLE word, and in this context, the above instruction is invalid. However,

```
1 CHAN -100 PITCH 5 WAVE 100 AMP
```

can be entered as

```
1CHAN -100PITCH 5WAVE 100AMP
```

or

```
1CHAN       -100  PITCH    5WAVE100AMP
```

and so on.

This chapter should serve to give you a small taste of the Music 500's capabilities. The next chapter will take you through many more facilities, and answer the questions you probably already have after after reaching this point!

TUTORIAL SECTION

5 Introduction

Starting the language

Having loaded the language from cassette, - see chapter 2 - you should see the following startup message

```
AMPLE Model BCE Version 1.0
(C) 1984 Hybrid Technology
 %_
```

Now type

"preset1" LOAD RUN

and press the PLAY button on your cassette player. When loaded, (the **%** prompt reappears), stop the cassette. This loads the preset wavforms and envelopes. Remember that if you press **BREAK** or type **AMPLE RETURN** you will need to load these in again.

Instruments, parts and players

If you imagine a group of people playing musical instruments, they consist of:

Players
Parts
Instruments

Each person is a player who may be playing a different set of notes at different times to the other players. He will not necessarily be playing the same sound continuously, eg a violinist may be using his bow sometimes, and plucking the strings at other times.

The Music 500 works in the same way. Starting from the bottom level up,

A number of **CHAN**nels are used to build a particular instrument

That instrument is given to a **VOICE**

One **VOICE** - or more - is given to a **PLAYER**

When the Music 500 is first switched on, various decisions are automatically made for you. You have at your disposal a **PLAYER** with one **VOICE** having two **CHANs**. This saves you the effort of having to define these before you can start producing sounds. This chapter deals with simple sounds, and will only be using the 'default' settings of one **PLAYER**, one **VOICE** and two **CHANnels**, but the explanations which follow are based on your awareness of these categories.

Key functions

Here is a brief description of the effect and uses of keys on the BBC Microcomputer keyboard.

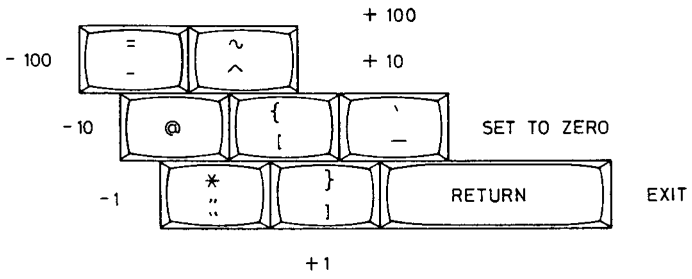
RETURN Executes the line just entered
ESCAPE Silences the Music 500

DELETE Deletes the character before the cursor
COPY & cursor keys Line editing

CTRL U Scraps the current line before **RETURN** is pressed

f0 - f9 Should be used to store commonly used commands (see the *BBC* Microcomputer User Guide for details)

BREAK Returns computer to BASIC. *Don't hit BREAK!*



SCANNING KEYS

6 Playing some notes - II

The section entitled *Playing some notes* in the previous chapter gives a detailed account of the up/down note system, rests, fixing the pitch, varying the note length, ties, sharps and flats. If you have not grasped these points, please read the section again.

In addition to the above, here are some more music words.

Moving octaves relative to last note

To move either up or down an octave relative to the previous note, use the words > and <. For example, here is a run of three notes repeated over three octaves using the : pitch fixing word

```
15, -1:Edc 0:Edc 1:Edc ↑
```

Now the same thing using the > word

```
15, -1:Edc >Edc >Edc ↑
```

The same applies to the < (relative octave down) word

```
1:cDE 0:cDE -1:cDE ↑
```

can be written

```
1:CDE <cDE <cDE ↑
```

To get a multiple octave jump, use >>, or <<< etc.

Changing the tempo

The tempo of a complete piece of music can be increased or slowed down by the **TEMPO** word. The advantages of changing the tempo will be more apparent after a little practice. Individual notes can be have their length changed relative to each other, as in 'Frere Jacques' below

```
48,-2:CDEccDEcEF 96,G 48,eF 96,G 24,GAgf 48,ec 24,GAgf
48,eccgC/CgC/↑
```

If you wanted to speed this up, you could change all the note lengths, but this be very tedious. The **TEMPO** word is used instead.

The initial value of **TEMPO** is **1000**, and dividing by two will double the speed as multiplying by two will halve the speed. Maximum and minimum values are **26** to **65535**. So, in our example, to increase the speed a little,

```
700 TEMPO
48,-2:CDEccDEcEF 96,G 48,eF 96,G 24,GAgf 48,ec 24,GAgf
48,eccgC/CgC/↑
```

7 Storing the words

Up to now, whatever words you have typed in have been executed immediately after pressing **RETURN.** Not only this, but after you have executed the command, you have to type it in again (or use the cursor and **COPY** keys if it's still on the screen) to repeat it.

Storing a sequence of words is done by starting each line you type in with line number *followed by a fullstop.* So you can enter 'Frere Jacques' as follows (where you break the line is arbitrary)

```
10.SCORE % Frere Jacques
20.48,-2:CDEccDEc
30.EF96,G 48,eF 96,G
40.24,GAgf48,ec 24,GAgf 48,ec
50.cgC/CgC/↑
```

Notice the **%** sign half way through line 10., which will cause the rest of the line to be ignored by AMPLE, allowing comments to be included.

Listing the stored words

The effect of the line numbers is to stop each line from being executed after you have entered it, and to store the complete contents inside the computer.

To prove this, type

```
LIST
```

and the above lines are displayed, showing that they are indeed stored inside the computer. Remember that nothing has been executed yet, the listing above is simply text stored somewhere inside the computer.

Executing the stored words

To execute it, type

```
RUN
```

This has the same effect as if you had typed in the words directly, and the tune plays. To play the tune again, type **RUN RETURN** again.

Editing the listing

To add a new line, decide where in the list of words you want the new command to be executed, and choose a spare line number. For example, to insert another comment on a line by itself

```
15. % An extra comment
```

Now **LIST** again, and the result should be

```
10.SCORE %Frere Jacques
15.% An extra comment
20.48,-2:CDEccDEc
30.EF96,G 48,eF 96,G
40.24,GAgf48,ec 24,GAgf 48,ec
50.cgC/CgC/↑
```

To make any changes within a line - retype the line or, type **LIST RETURN**, and use the cursor and **COPY** keys to locate and copy the line(s) and make changes in the process.

If you run out of spare 'in-between' line numbers, type

```
REN
```

and the line numbers will be renumbered **10. 20. 30. 40.** and so on.

Line can be completely deleted by typing in the line number *including the fullstop* and pressing **RETURN**. (This actually replaces the existing line with no text.)

Text only!

Remember that until you type **RUN RETURN**, any changes you have made are still merely changes to the text stored in the computer.

Erasing stored text

The word **CLEAR** is used to clear the stored text from memory, so you can start a new listing without having to delete all the old line numbers one by one.

Loading and saving listings

Having built up a sound, a note sequence or both, it can be saved to cassette. This allows you to build up a library of complete compositions, interesting instruments to be used in future compositions, (and for the more advanced - your own waveforms and envelopes). The command to do this takes the form

"name" **SAVE**

Where   name   is any word which obeys the rules for filenames on the cassette filing system - if in doubt refer to your *BBC Microcomputer User Guide*.

Loading a file off cassette is done by typing

"name" **LOAD**

and this will replace any listing currently in the computer with the loaded one.

Using the example below,

```
10. SCORE %Frere Jacques
15. % An extra comment
20. 48,-2:CDEccDEc
30. EF96,G 48,eF 96,G
40. 24,GAgf48,ec 24,GAgf 48,ec
50. cgC/CgC/↑
```

Type

"frere" SAVE

and record it onto cassette in the normal way. Type **CLEAR RETURN to remove it from the computer memory, and LIST RETURN** to prove that it no longer exists. Now rewind the tape to the beginning of where you loaded the file called **frere** and type

"frere" LOAD

and press PLAY. When the **%** prompt reappears, stop the tape, and **LIST** it. To play the piece, (it's stored in the computer as non-executed text - remember?), **RUN** it.

8 Changing the sound

Starting up

In the previous chapter we only used the default sound for playing notes. To start off creating some new sounds, type

SOUND

and press **RETURN.** This sets everything back to when you first loaded AMPLE into the computer, regardless of anything you may have typed in so far.

Initial conditions

When you first start up, or after typing **AMPLE**, you have by default

- One voice

- Two channels, only one of which is switched on at maximum volume

Any sound words you type now in will affect voice number one, and channel number one.

To provide a continuous sound, voice number one has to be put on 'standby' for being subsequently switched on, and both actions are carried out by typing

1 VOICE SOUND ON GATE

The **1 VOICE** in this case isn't necessary, because the startup condition is a single voice, so **1 VOICE** is automatically implied.

You will now hear a tone. (**ON GATE** is used by music words to make notes sound.)

Simple sound words

Changing the pitch

Now enter

1 CHAN 100 PITCH

The pitch should jump up. Its normal value is **0**, or middle C. Again, the music words use this to set the pitch of the notes. Try different values of **PITCH** - positive and negative. The limits are -1024 to 1023, and a complete octave is 192 units.

Alternatively, you can continuously range up and down the **PITCH** numbers by using another sound word - **SCAN**. Type

SCAN PITCH

(**1 CHAN** *is implied*.) The pitch resets itself to middle C (**PITCH** no. **0**), and the number is displayed on the line below. To scan up and down in increments of one unit, use the 'colon' and 'close square bracket' keys. To scan in increments of 10 units, use the '@' and 'open square bracket' keys. To finish scanning, and leave the pitch at the value displayed (and heard), press **RETURN**. Please refer to the Glossary for the **SCAN** keys.

Changing the tone

Changing the tone is done by using the **WAVE** word. The tone is governed by the waveform of the sound, and there are fourteen ready-made **WAVE**s, and the one you can hear at the moment is number **0**. To change the waveform, type

3 WAVE

The sound changes tone to **WAVE** number three. **SCAN** can be used again for speed, so type

SCAN WAVE

The original **WAVE**, number **0** is selected, but using the 'colon' and 'close square bracket' keys shows you what the others sound like. Again, press **RETURN** to leave scanning. For best effects, keep the pitch number at around -200 while you scan through the **WAVE**s.

Stereo positioning

The stereo position is currently in the middle, but it can be shifted to either side by the **POS** word. The number entered before **POS** determines the position, from **-3** (extreme left) to **0** (middle) to **3** (extreme right. The **POS** word can also be scanned with **SCAN**. Try typing

-3 POS

2 POS

etc.

Volume

To alter the volume of the sound, use **AMP**. This takes numbers from 0 to 128 (loudest), although the useful range starts at about **30** upwards. Thus

90 AMP

reduces the volume, and

128 AMP

restores the volume to the normal maximum level. **SCAN** works with **AMP** as well.

Two sounds together

Up to now, you have only been using one 'channel' of sound on voice number one. There is another 'channel' available but not doing anything. The channel you're using, number one, is normally already turned on and at maximum volume. The second channel isn't. So type

2 CHAN ON GATE 128 AMP

This turns on channel 2, and sets the volume to maximum. You will now hear channel 2 (waveform number 0 of course), and in the centre stereo position. **Note that any words you type now will affect channel 2 only.** To change anything on channel 1, type

1 CHAN

first, followed by your instructions - either on the same line, or after **RETURN**.

Now let's start again, and with what you've done already, you can experiment with two channels and their effect on each other.
Type

```
SOUND
1 CHAN 3 WAVE
-400 PITCH ON GATE
2 CHAN ON GATE 128 AMP
5 WAVE -400 PITCH
```

What you hear now is two channels playing at exactly the same pitch, same volume and in the middle stereo position. Anything you type in now will affect the second channel until you enter **1 CHAN**. So type

```
3 POS
1 CHAN -3 POS
```

This shifts channel 2 to the right speaker, and channel 1 to the left speaker.
Now restore them to the middle again by typing

```
0 POS
2 CHAN 0 POS
```

Shifting the frequency slightly

The word **OFFSET** works rather like the **PITCH** word, but allows you to shift the frequency of a channel very very slightly.
Shifting the frequency of one channel in relation to another channel when both are originally at the same pitch, produces a very interesting effect known as 'phasing'. Type

```
200 OFFSET
```

to offset the frequency of channel 2. If you want to find out what different values of offset do, type

```
SCAN OFFSET
```

and use the up/down keys to continuously vary the value.

Try changing the pitch and waveforms of either channel.

Shifting the pitch

Every channel can be shifted in pitch by using the **SHIFT** word.
For example

```
2 CHAN 192 SHIFT
```

Type **SCORE** **RETURN** and play some notes.

NB: **SHIFT** can be used to shift the pitch of musical notes, and can be used with the voice, so

Group words

In the last example above, you may have realised another powerful feature of the Music 500, which is that the **PITCH** **POS** **SHIFT** **AMP** words can be applied to the complete instrument as a whole, as well as to individual channels - by giving commands to the voice.

Storing sounds

Sound words can be stored in exactly the same way as music, so build a sound you like and enter it with line numbers as you did with Frere Jacques, and store it on cassette.

Playing tunes with the new sound

Now that you've created a more complex sound, type

SCORE

and enter a sequence of notes to try out the new sound. You can make any changes you like to the voice or channels by typing them in.

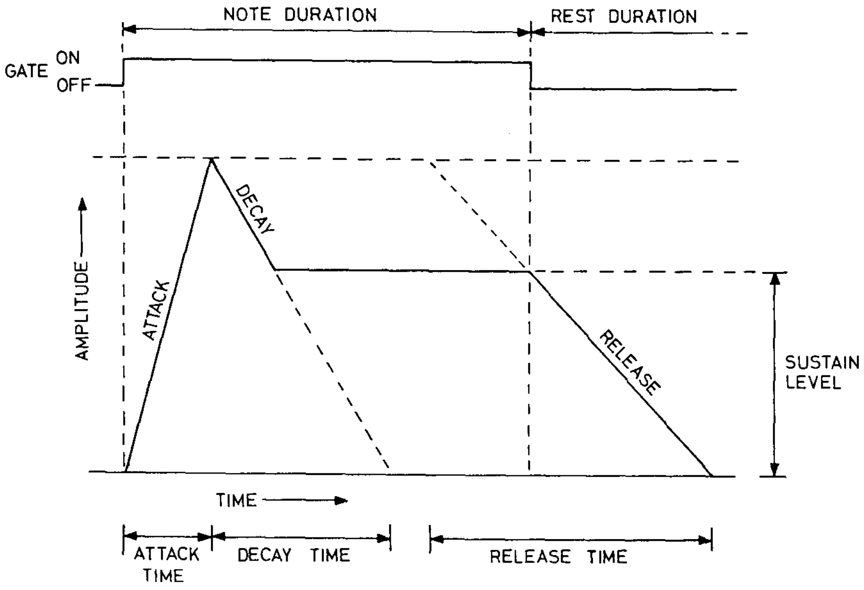
Using envelopes

You can create your own envelopes to change the amplitude (volume) or pitch within each note.

Amplitude Envelopes

The amplitude envelope is the shape of the note - how fast it builds up when the note starts, whether it dies away while the note is held, etc. Useful shapes include organ-like, where the amplitude is constant throughout the note, and percussive, where it decays to zero immediately, giving a struck or plucked effect.

Many different shapes can be created by varying a basic shape called ADSR. This stands for Attack Decay Sustain Release; the four parts that make it up. It looks like this:



The attack segment is the build-up at the start of the note. It follows on immediately into the decay segment, which falls to a sustain level. When the note goes off, the release section takes the amplitude down to zero.

To create an ADSR, type:

1 EMOD ADSR

EMOD sets which one of the 13 envelopes stores is to be modified; in this case we are going to put the new envelope in number 1.

To play the envelope, type

SOUND 1 AENV SCORE

followed by some notes. We can now vary the parameters of the ADSR. Lets start with the sustain level:

0 SUSTAIN

This makes the amplitude die away to zero when the note is held, like a piano. The time it takes to do so is controlled by **DECAY**. We can **SCAN** this along with some notes to show the effect:

48, SCAN DECAY C↑

Press → to increase the decay time and play the note. The number is the decay time in centiseconds.

ATTACK sets the build-up time of a note after a rest. To make a soft start to each note, try

20 ATTACK c↑G↑c↑G↑

The release section is only heard when a note is followed by a rest. It works like decay, but starts form the sustain level, or whatever other level the envelope is at when the note ends. To hear its effect, raise the sustain level to maximum by

127 SUSTAIN

and type

200 RELEASE C↑

You will hear the note die away over the period of 2 seconds.

Envelope definitions

Like sound words working on the current channel, envelope words work on the current envelope set by **EMOD**. This is independent of the envelope used by a channel, set by AENV, so you can have a group of channels using different envelopes, and switch between them to make modifications.

ADSR is a bit like **SOUND**, setting the envelope to an initial state. Some of the preset envelopes are not made with ADSR so do not try to modify them, and be sure to set one up with **ADSR** if you want to define your own.

To use a new envelope in a piece, you put the **ADSR** words that create it into a word to make an envelope definition. For example

```
"dong" ← ADSR 100 DECAY 90 SUSTAIN →
```

This is then used to set up a particular numbered envelope, which can be selected as normal for any channel.

```
3 EMOD dong        % create dong envelope as number 3
1 CHAN 3 AENV      % use it on channel 1
```

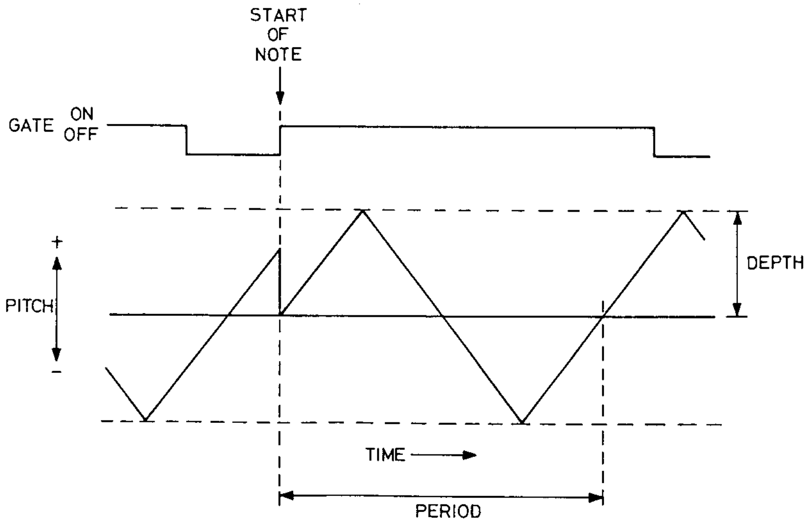
Pitch Envelopes

Pitch envelopes vary the pitch of a sound automatically, adding on to the pitch set by the sound words **PITCH** and **SHIFT** to create variations within notes.

A simple kind of pitch envelope is a repeating triangle shape which slopes up and down between two levels. **CYCLE** sets this up in the same way as **ADSR**, but rather than have additional words to set its parameters, it takes two numbers; the period of the cycle and the depth. Here's what a **CYCLE** envelope looks like:

Each of the waveforms can be changed by calling up one of the ??? predefined pitch or amplitude envelopes. For example, try this

(!!! INSERT **CYCLE** DIAGRAM !!!)



The cycle restarts on each gate on (each note start).

Try out this example:

```
2 EMOD  500 50 CYCLE
```

It programs envelope 2 to a cycle shape with a period of 500cs (5 seconds) and a distance of 50 pitch units above and below the normal pitch. The result is a wailing siren-type sound:

```
SOUND 2 PENV ON GATE
```

You can try playing music using this envelope if you like!

ON EBIG amplifies the pitch envelope by a factor of four, so that within **CYCLE**'s maximum depth of **127**, you can produce very wide sweeps. **OFF EBIG** returns to the normal scale factor.

A cycling pitch envelope much more suited to music is vibrato, examples of which are amongst the preset envelopes. Using **CYCLE**, you can specify the period and depth precisely. Here's an example of two vibrato's used together:

```
1 EMOD  32 3 CYCLE
2 EMOD  36 2 CYCLE

"chorus" ← SOUND 1 CHAN 1 PENV 2 CHAN 2 PENV 128 AMP →

chorus SCORE gDcf
```

They have slightly different depths and periods, so the two waveforms shift around relative to each other, producing a chorus effect.

Summary

The ESCAPE key

Pressing **ESCAPE** will immediately silence the Music 500. However, it is better practice to end a sequence of notes with a ↑.

SOUND and SCORE

Entering either **SOUND** or **SCORE** restores the sound type, and note lengths to the standard 'startup' values. To think of **SOUND** and **SCORE** as being two independant 'states' is mistaken. Once both words have been entered, ie

```
SOUND ON GATE
```

to generate a tone to monitor the building of a new sound, and

```
SCORE
```

to enable you to play notes, you can freely interchange between changing the sound and entering notes, for example

```
SOUND ON GATE
2 CHAN ON GATE 128 AMP 20 OFFSET
ALL CHAN 4 WAVE
SCAN PITCH
.
.
.
.
SCORE
-2:24,CEDFEG↑
1CHAN -3 POS
2 CHAN 3 POS
-1:24,CEDFEG↑
```

How the words are obeyed

You will have noticed by now that the Music 500 obeys an instruction by adding its effect to what has gone before. For example

```
2 CHAN -100 PITCH
2 CHAN 5 WAVE
2 CHAN 100 AMP
```

is the same as

```
2 CHAN -100 PITCH
5 WAVE
100 AMP
```

is the same as

```
2 CHAN -100 PITCH 5 WAVE 100 AMP
```

because having specified channel number 2, all following instructions will automatically be applied to channel 2 until you specify a different channel number.

The order within an instruction

You will also have noticed that you enter the number first, and the instruction the number refers to second. This is true for all AMPLE instructions, whether the particular value is a number or a word.

All AMPLE words are always entered in upper case, so **CHAN** cannot be entered as **Chan** or **chan**.

Spaces

Mostly, spaces can be inserted into what you type, or left out. Their main use is to make the displayed text readable. The only exception to this is where two AMPLE words without a separating space may make a third word, for example

```
ALL CHAN SCAN PITCH
```

will allow you to change the pitch on all the channels currently in use, but

```
ALL CHANSCAN PITCH
```

will give an error message, because **CHANS** is an AMPLE word, and in this context, the above instruction is invalid. However,

```
2 CHAN -100 PITCH 5 WAVE 100 AMP
```

can be entered as

```
2CHAN -100PITCH 5WAVE 100AMP
```

or

```
2CHAN       -100  PITCH     5WAVE100AMP
```

and so on.

9 Defining words

Introduction

In chapter 7, we discussed how to store a sequence of words in the computer's memory, then save it to cassette. Another very powerful feature of AMPLE is the ability to define your own words, and store them away in a library of your own definitions. The advantages of defining your own words as opposed to the 'line numbering' system you have been using are as follows

- more than one word can be defined and held in memory
- user defined words can be called from within other user defined words
- each user defined word is stored not as text, but like a standard AMPLE word, ie ready for immediate execution

Words can be defined to consist of note sequences, instruments, your own wave or envelope definitions etc.

Defining one line words

First type **SOUND RETURN** to reset everything. Now type

"scale" ← SCORE -1:CDEFGABC↑→

This defines the scale of C major as **scale**, and the definition begins after the **←** and ends before the **→**.

If you type **SHOW**

The word **scale** is displayed. To play it, type

SOUND scale

Now define another word to hold a sound. Type

"tone"←SOUND 1 CHAN 2 WAVE 2 CHAN 128 AMP 2 WAVE 20 OFFSET→

This defines the sound as **tone**, and the definition begins after the **←** and ends before the **→**. Type **SHOW RETURN** again, and notice that you now have two words, **scale** and **tone**. Now type

```
tone scale
```

to get the full effect. Now try defining more sounds, and use each of them with **scale**.

Defining words with line numbers

Many word definitions won't work unless they are entered as separate lines. This is where we can go back to the line numbering technique. Remember that *everything* must be line numbered, including the name of the word. So for example, our Frere Jacques piece could be entered as

```
5."fj" ←
10. SCORE %Frere Jacques
15. % An extra comment
20. 48,-2:CDEccDEc
30. EF96,G 48,eF 96,G
40. 24,GAgf48,ec 24,GAgf 48,ec
50. cgC/CgC/↑
60. →
```

Line 5. contains the name of the word, and the ← character states that a word definition called "**fj**" is to be defined. Line 60. contains the → chanaracter which ends the definition.

Because any definition is stored ready for execution, and *not* as text, the lines must first be translated from text into the corresponding instructions. So type **RUN RETURN** to do so.

Now type

```
SHOW
```

This word displays all the current definitions in the library, and as you can see, it contains **fj**.

To play **fj**, type

```
SOUND fj
```

or replace **SOUND** with one of your user defined sounds.

Editing and deleting user defined words

Editing

First type **CLEAR RETURN** to demonstrate the following example.

To display the contents of a user defined word on the screen in order to edit it, AMPLE must first translate it back into text again. It does so with the **EDIT** command. So to display **fj** on the screen (**LIST** won't work because you've just **CLEAR**ed the listing!) type

"fj" EDIT

The text version of **fj** is displayed on the screen for you to change as you want. As before, to complete the redefinition, you *must* **RUN** it to update the previous definition.

Deleting

To delete a user defined word (which also erases it from the library), type

"word" DELETE

This may not seem that important, but its usefulness will become clear in the next section on saving and loading.

Saving user defined words

When you have defined all the words necessary to your composition, you will want to save them on cassette. Think up a suitable name for the composition, say 'Recital', and type

"recital"SAVE

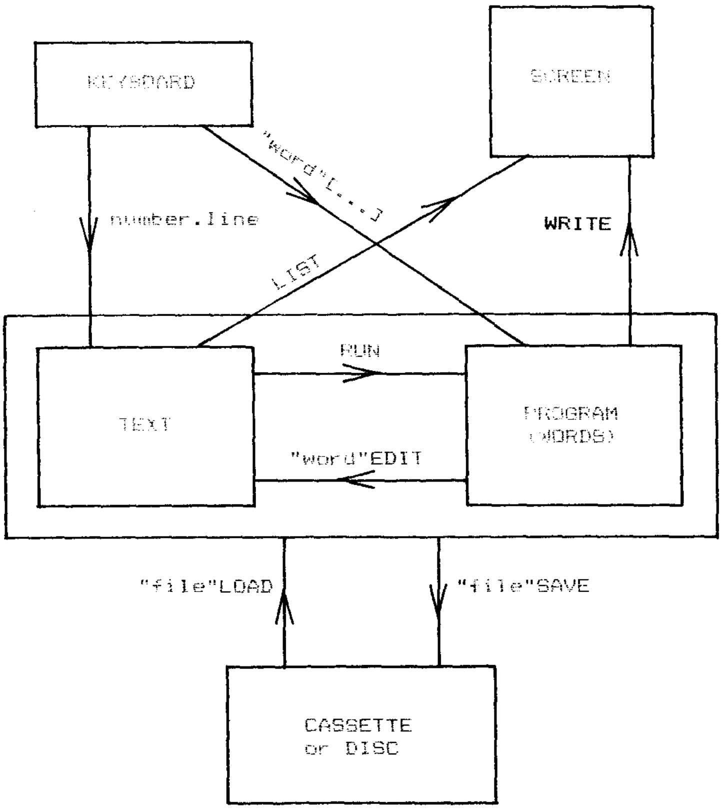
Find a suitable place on your cassette, and record. All the user defined words in the library will be saved. If you have some redundant words in the library, you might want to delete them first, before saving the library - see the section called *Deleting* above.

Loading user defined words

To load a composition back into the computer which contains user define words, type

"recital"LOAD

(using 'Recital' as an example). This will load into the library all the words associated with 'Recital'.



Diagrammatical summary of this chapter

Advanced AMPLE

10 Advanced features

Key Signatures

The key signature is a set of modifications of the pitches of the notes to suit different keys.

The words K(and)K set the key signature. They are always used as a pair - each K(must be followed by a)K. To set a sharp or flat in the key signature for a particular note, just include the note between K(and)K with the sharp or flat before it as normal. The note doesn't play, but merely causes its modification to be stored and applied to all same-lettered notes in the music that follows.

Here are some examples for common keys:

```
 C major and A minor K( )K
 G major and E minor K( +F )K
 D major and B minor K( +F +C )K
 A major and F# minor K( +F +C +G )K

 F major and D minor K( -B )K
 Bb major and G minor K( -B -E )K
 Eb major and C minor K( -B -E -A )K
```

SCORE sets the key signature to no flats and no sharps, that is, K( )K, so for parts in C major there is no need to set the key signature.

Naturalising Notes

The = word represents a natural sign. Used before a note, it removes the effect of the key signature so it plays at its unmodified pitch. It works on the next note only, so must be put before every note that needs to be naturalised.

+ and - have the same effect as = ; they override the key signature's modification before applying their own.

Using bar lines

Bar lines can be included in music to make it more readable and to check that each bar is has the correct number of notes of the right length. Bar lines has no effect on the sound of the music.

The checking function is very useful for all but the smallest pieces, detecting missing and extra notes, and notes of the wrong length, which could otherwise be difficult to locate.

The ¦ word represents a bar line. Here's an example of its use:

```
 24,gfGf 48,c/ ¦ 24,GfGf 48,dc ¦ 24,bC 48,//D ¦
```

Bar checking is initially turned off after SCORE. To turn it on, you must set the length of the bar with BAR. Once this is done, any bar line which finds that the total of notes lengths since the last bar line (or SCORE) is different, will cause the 'Bad bar' error. The bar length for the example above is 192, corresponding to 4/4 time with a crotchet length of 48.

```
SCORE 192 BAR 24,gfGf 48,c/ ¦ 24,GfGf 48,dc ¦ 24,bC 48,//D ¦
```

For a 3/4 time signature, 144 BAR would be used.

Note that you should not put a bar line at the start of a part, after SCORE.

To disable bar checking, set the length to 0, i.e.

```
0 BAR
```

This is a good idea if you are playing barred phrases at the keyboard, where the first or last bar may be intentionally incomplete.

Basic polyphony

Voices

The Music 500 has eight voices, meaning that you can play up to eight notes at the same time. Normally just one voice is available, but you can make more available usimg the word VOICES. It takes a single number which is the number of voices to be assigned, for example

```
3 VOICES
```

to get three voices. If some voices have been used up elsewhere, then there may not be three available and the 'Too many voices' error will appear. This can happen if you have just run a program that uses dynamic players; one of the demonstration programs for example. To make sure that there are no voices used by dynamic players, enter STOP.

You can switch between the three voices using VOICE. It takes the number of the voice, which must have been assigned, and selects it so that future sound commands will go to it and not the other voices. With three voices assigned, the following will play a chord

```
 1 VOICE SOUND ON GATE
       2 VOICE SOUND 112 PITCH ON GATE
 3 VOICE SOUND 192 PITCH ON GATE
```

Each newly assigned voice must be set up with SOUND before it is used. As far as sound is concerned, the three voices are entirely separate and you can set each one up with a completely different instrument.

Playing chords

We can control a group of voices using music words, to play chords for example. The voices must have been assigned and set up with VOICES and VOICE. Music words select the voice for themselves, overriding the voice selected by VOICE.

After SCORE, normal notes will play on voice 1 and the other voices will be quiet. To play a note on voice 2, we enclose it in round brackets, for example

```
 SCORE c(G) c(A) F(A) d(G) ↑(↑)
```

The notes in brackets play in parallel with the previous note outside the brackets, lasting for the same length of time. Notice the bracketed rest which stops the last note on voice 2.

Each sucessive note in a bracketed group plays on another voice, so we can add a note to each group to play three note chords:

```
 SCORE c(GE) c(AE) F(AC) d(GC) ↑(↑↑)
```

Notes inside groups have no effect on the pitch of notes outside, so the first notes of each chord play at the same pitch as they would have done had there been no parallel notes. However, the first note in a group has its pitch determined relative to the previous note in the normal way, and the scheme applies as usual inside the group. This means that, in our example, the notes in each chord go up from the main note, since they are all upper case. It would be quite possible to make them go down, but it is usual to have the main note at the bottom and the parallel notes above it.

Ties can be used in chords to leave a voice unchanged, carrying through its note from the last chord. Here's our example with some of the notes suspended from chord to chord:

```
 SCORE c(G/) c(A/) F(/C) d(G/) ↑(↑↑)
```

A note played on one voice continues to play until another note, or rest, plays on the same voice. This means that we can start a note in parallel and then play independently on the voice 1 with unbracketed notes, for example:

```
 SCORE C(g) DEdc-b(f) CDc-ba(e) ^(^)
```

The G F and E are sustained underneath the main tune.

Selecting voices

In some cases like the last example, where there are more than two voices playing but not as simple chords, it is convenient to select the voice explicitly. This can be done with the ';' word, which takes the number of the voice to be used for the following notes.

';' is not the same as VOICE - it sets the voice on which music will play. In fact, the current sound voice is set to this, using VOICE, when the music words play their sounds, but in most situations this happens well out of sight.

Here's the last example written with ';' :

```
 SCORE C 2;g 1;DEdc-b 2;f 1;CDc-ba 2;e 1;^ 2;^
```

This is exactly equivalent to the group example.

Assigning channels

The Music 500 has 16 sound generating channels. Each voice initially has two channels assigned to it, meaning that the the sound of the instrument can be made up from two simultaneous sounds.

CHANS is used to assign more channels to a voice to make more complicated instrment sounds. It takes an even number in the range 1 to 16, for example:

```
 4 CHANS
```

to assign four channels to the curent voice.

If there are not enough channels spare, the 'Too may channels' error wil be produced. In the case of 4 CHANS, this would happen if there were eight voices, since each would normally have two channels.

To free all extra channels, type STOP to free voices (and their channels) owned by players, followed by 1 VOICES. With only one voice in existence, you can have all sixteen channels on it.

The voice's channels are numbered from 1 upwards, and are selected using CHAN as normal, for example:

```
 4 CHANS SOUND      % sound sets up all channels on voice
 2 CHAN 128 AMP  112 OFFSET % up fifth
 3 CHAN 128 AMP -192 OFFSET % down octave
 4 CHAN 128 AMP  -80 OFFSET % down to fifth
```

The four channels play a simple chord around the note pitch.

If this was made into an instrument definition word, the 4 CHANS would be included in the word because it is the instrument that should say how many channels it needs to make its sound play.

Changing sounds in music

You can use individual sound words and instrument definitions in the middle of music to change the sound 'on-the-fly'.

To make the change of sound happen at the start of the note following it, the @ word is used before it, for example:

```
 48,aaEE | @ fuzzins  aaEE |  % change to fuzzins

 @ -3 POS riff @ 3 POS riff     % play on left, then right
```

The @ completes the duration of the previous musical event. It is only required before the first of a sequence of sound words.

Defining waveforms

The Music 500 can store 13 waveforms. You can create your own waveforms and put them in any of the 13 waveform stores, from where they can used by channels.

Harmonic synthesis

AMPLE allows you to define a new waveform by specifying the strength of its first 16 harmonics. There are three stages to defining a waveform harmonically:

1 select a waveform using WMOD
2 set the amplitudes of the harmonics using WH!
3 convert and copy the waveform using WHG and WGC

These stages can be carried out by typing words in directly to experiment, but when a waveform has been finalised, stages 2 and 3 are usually carried out by a waveform definition. Analogous to an instrument definition, this is a word that contains the instructions to make a particular waveform, which is then used independently.

Here's a typical harmonic waveform definition:

```
"reedy" ←
WZERO    % set all harmonics to zero
                % set harmonic amplitudes:
 90 1 WH!       % medium fundamental 127 3 WH!       % strong
third
 80 5 WH!       % medium fifth 100 7 WH!       % strong seventh

 60 9 WH!       % weak ninth WHG WGC →       % convert, and
copy to current waveform
```

Harmonic definitions use the harmonic waveform buffer - a temporary store in which the set of harmonic amplitudes is put together. It is not connected with the current waveform or any other particular waveform.

WZERO clears all the harmonics to zero.

WH! sets the strength of a harmonic. It takes the relative amplitude in the range 0 to 127, followed by the number of the harmonic. After a list of WH!'s, WHG converts the set of harmonics into geometric form, and WGC copies this the current waveform so that the new definition can be used.

The current waveform is totally independent of the waveform selected for a channel by WAVE. Similar to the current envelope, it is the waveform that is being modified, and is set by WMOD before the definition is used. You can use any one of waveforms 1 to 13 for the new definition. Here's our example definition used on waveform 8:

```
8 WMOD reedy
```

Waveform 8 is now 'reedy' and can be selected as normal by any channel, using WAVE.

Here are some points to bear in mind when composing waveforms harmonically:

1Natural sounds often have a strong fundamental (first harmonic) and harmonics which die away to zero as they get higher.

2The balance of low to high harmonics determines the brightness - the stronger the high harmonics, the brighter the tone.

3Strong odd harmonics give a reedy, organ-like tone.

4If the fundamental and low harmonics are weaker than higher harmonics, the tone is thin and weedy.

5The fundamental can be left out entirely and a few strong harmonics arrranged sparsely for odd, seemingly non-harmonic tones.

6Waveforms for use at low pitch are normally brighter than those for use at high pitch.

7Waveforms with strong high harmonics can become distorted and noisy when used at high pitch.

Geometric synthesis

You can also create waveforms geometrically, by specifying the shape of the waveform point-by-point. Relating the geometric shape of a waveform to its sound is much more difficult than relating its harmonic shape, so geometric creation more suited to special waveforms. These include common waveform shapes such as triangle, ramp and pulse, and those that cannot be created harmonically, such as random waveforms for creating noise sounds.

Making waveforms geometrically requires some understanding of AMPLE's programming words, since the point values are invariably created by short programs. The geometric waveform buffer is used to hold the points of the waveform as they are calculated, and when the definition is complete, its contents are copied to the current waveform, set by WMOD.

WG! is used to fill the buffer. It takes the value to be written, in the range -127 to +127, followed by the number of the point, in the range 1 to 128.

Here's a simple example:

```
"randwave" ← 128 FOR( RAND? INDEX WG! )FOR WGC →
```

The loop goes around all points, writing a random value into each. WGC copies the contents of the geometric buffer to the current waveform, which would be set before 'randwave' is used, for example:

```
5 WMOD randwave
```

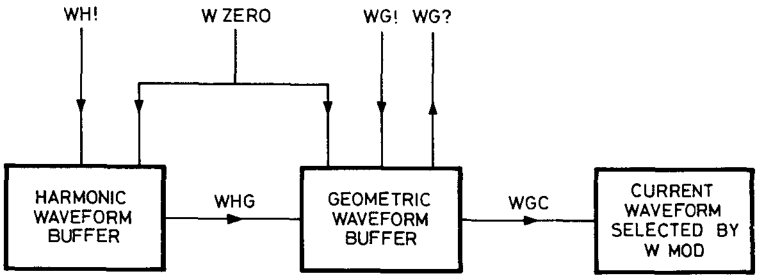
Here's another example; a waveform definition that produces a pulse wave of the specified width.

```
"pulse" ← % widthnumber -> .  0 <= width <= 127 128 FOR( #11
INDEX #<                     % compare index with width
  IF( 127 )ELSE( -127 )IF    % leave max or min, and
  INDEX WG!                  % write into point
    )FOR WGC #2 →                      % convert and discard width
```

Like the random wave, pulse waves are very rich and are best used at low pitches.

Waveform processing

Once a waveform has been created, either geometrically or harmonically, you can modify it before it is copied to the current waveform. Here's how the waveform creation system looks as a whole:



The complement of WG! is WG?, which reads values from the geometric buffer. It takes a point number and returns the value at that point, allowing you to process the waveform in the buffer.

Clipping is a simple and effective example of waveform processing. It cuts off the positive and negative peaks of the waveform to make a brighter tone. To avoid making the waveform quieter, the limits are set at + and - 127, and the waveform amplified within them.

```
 "clip" ←                 % amplify and clip waveform
 128 FOR( INDEX WG?   % read point
    2 #*          % multiply by two
          127 MIN      % positive limit
    -127 MAX      % negative limit
     INDEX WG!     % write point
     )FOR →
```

The 'clip' processing word works on the existing contents of the geometric buffer. First the input waveform is setup, then clip is used, and then the buffer is copied to the current waveform.

```
"evenwave" ←              % waveform definition
 WZERO
 127 1 WH! 127 2 WH! 100 4 WH!
 WHG →                    % harmonic -> geometric, but don't copy

 1 WMOD evenwave          % create waveform
 clip                     % process geometric
 WGC                      % copy to current
```

You could clip the waveform more than once for a very harsh tone, or modify clip to give unsymmetrical clipping by adding an offset to the input point values.

Another useful process is low-pass filtering - gradually removing harmonics above a certain number. The simplest filter algorithm averages adjacent points, smoothing sharp corners. This is effective at removing unwanted buzzyness in harsh waveforms such as the clipped ones in the last example.

```
 "lpf" ←
 128 WG?                     % keep first point
 127 FOR( INDEX WG?  % get point
    INDEX 1 #+ WG? % and next point
    #+ 2 #/ #2    % average
          INDEX 1 #+ WG! % put in next point
     )FOR
 1 WG? #+ 2 #/ #2 1 WG!   % do last point with saved first point
 →

 1 WMOD evenwave       % create waveform
 clip          % process geometric
 lpf lpf lpf           % filter three times
 WGC                   % copy to current
```

Finally, here's a word to display the points numerically; useful for debugging processing words.

```
"points" <-
 128 FOR( INDEX WG? &FF AND % get point value
         &$STR 4 $PAD $OUT % print number in field of four
     )FOR NL ->
```

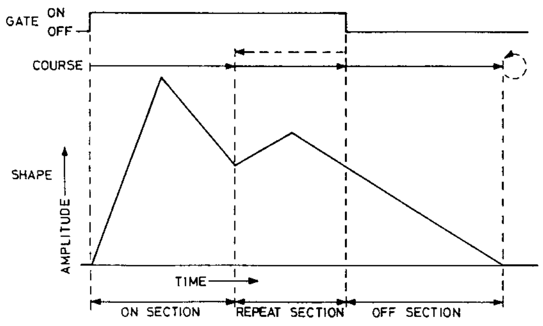
Advanced Envelopes

ADSR and CYCLE allow you to vary the parameters of two useful shapes of envelope, but you can make different shapes entirely using the segment envelope words. These work on the current envelope, set by EMOD, creating envelopes that can be used for pitch or amplitude.

The basic unit of an envelope is the segment - a straight-line portion between two points. Just a few segments are enough to make a wide range of useful shapes.

The segments are grouped together in three sections which act at at different times as the envelope plays.

The On section is done at the start of the note, when the gate goes on. It corresponds to the attack/decay part of an ADSR envelope. When it finishes, the Repeat section starts and continues while the note is playing. This is used for repeating effects like CYCLE. At the end of the note (when the gate goes off), the envelope enters the Off section. This corresponds to the release of an ADSR. When the off section finishes, the envelope value stays constant until the next gate.



The first step in defining an envelope is setting the number of segments in each section using ESECT. This takes the size of the on, repeat and off sections in that order, for example:

```
2 2 1 ESECT
```

These values are for the envelope in the example: two segments for each of the on and repeat sections, and one for off.

The next step is to program the segments. Each segment has two parameters; gradient and target level. The target level marks the end of the segment and the beginning of the next.

EGRAD sets the gradient as a fraction; the amount of time taken to go a certain number of units. Its arguments are the number of units, the time, and the segment number, in that order, for example

```
   1   10 1 EGRAD   % segment 1 goes up 1 unit per 10cs
 -32 100 2 EGRAD   % segment 2 goes down 32 units per 100cs (1s)
```

ELEV sets the target level for a particular segment:

```
   100 1 ELEV       % segment 1 stops at 100 units
   -32 2 ELEV       % segment 2 stops at -32 units
```

Here's the complete definition of a repeating amplitude envelope:

```
 "repeat" ←
  0 2 1 ESECT                     % two segments in repeat section
  127 40 1 EGRAD 127 1 ELEV       % up by 127 in 0.4s, stop at 127
 -127 10 2 EGRAD   0 2 ELEV       % down by 127 in 0.1s, stop at 0
 -127 10 3 EGRAD   0 3 ELEV       % fall to zero for off
 →
```

This creates a ramp shape that climbs to maximum quickly and falls to 0 slowly, repeating twice a second while the note is on. It is used after setting the current envelope with EMOD:

```
 6 EMOD repeat       % use envelope 6 for ramp envelope
 6 AENV              % select for current channel
 ON GATE             % start going
```

The on section has zero segments so the envelope goes straight into the repeat section when the gate goes on. The off segment takes the amplitude to zero when the gate goes off.

The first segment played after a gate is slightly special because it has no fixed starting level and picks up the envelope from the value it was left at. We could reset the envelope on each gate on by adding an on segment that went down to zero immediately, ensuring that the note started with a full cycle.

The next example shows how very shallow gradients can be used to hold the envelope value at a constant level for a certain period of time. It alternates the pitch between note pitch and a fifth above, sliding from one to the other.

```
 "fifthslide" ←              % envelope definition
 1 4 0 ESECT                    % use four repeat segments
   127 1 1 EGRAD    0 1 ELEV  % jump to normal pitch at start
   1 100 2 EGRAD    1 2 ELEV  % wait for 1s at lower pitch
 112  50 3 EGRAD 112 3 EGRAD % slide up over 0.5s
   1 100 4 EGRAD 113 4 ELEV  % wait for 1s at higher pitch -112
50 5 EGRAD    0 5 ELEV  % slide down over 0.5s →

5 EMOD fifthslide               % create 5 PENV
  % select SCORE 192,CDbC↑                % use
```

Because the off section has no segments, gate off's are ignored and the pitch effect continues during the release section.

Modulation

Combining channels

Modulation is a method of creating complex timbres by combining channels in pairs. A pair consists of an odd-numbered channel and the next highest even-numbered channel. For example, channels 1 and 2 are a pair.

There are three channel identifier words for use with CHAN to make use of channels in pairs easier. They are used as follows:

```
 ODD CHAN select all odd channels
 EVEN CHAN select all even channels
 num PAIR CHAN select the specified channel and the other
   channel of the same pair
```

Ring Modulation

Ring modulation is a technique that makes complex tones by combining two signals by multiplication. For each component in one signal, each component in the other creates two components in the result whose frequencies are the sum and difference of the original frequencies, producing complex and aperiodic sounds.

The odd channel produces the modulated waveform and its amplitude controls work in the normal way.

The modulating waveform is a two-state sign-only version of the even channel's signal. Its effect on the odd channel's waveform is to switch between normal and inverted, even within cycles.

The tone of the modulated signal depends on:

1 the modulated waveform
2 the modulating waveform
3 the frequency ratio (pitch interval)

The most important of the these is the frequency ratio, equivalent to a pitch interval between the channels. Different intervals give different tones.

If the interval is simple, for example unison or an octave up or down, the sound will be harmonic. This means that the components will heard as a harmonic series, typical of simple vibrating sound sources like strings and air columns. The sound will be bright, due to the high harmonics of the modulating waveform.

Adding small frequency offset (by OFFSET) causes the tone to change cyclicly, for example:

```
SOUND 1 CHAN ON RM 200 OFFSET ON GATE
```

More discordant intervals produce non-harmonic sounds. Very complex and distorted waveforms can result. Sweeping either pitch produces widely varying and completely un-natural tones.

Where the modulating waveforms are pseudo-random, the modulated waveform is noise-like. The characteristic of the noise are determined by the pitch and interval.

```
SOUND      % uses preset waveform 11 1 CHAN 11 WAVE ON RM ON GATE
2 CHAN 11 WAVE 3000 OFFSET % adjust offset for best effect
```

Synchronisation

Synchronisation is a technique that makes complex tones by a form of waveshape modulation. The modulated waveform is set to start of its cycle on each cycle of the modulating waveform. This forces the modulated waveform run at the modulating frequency, whereupon its own frequency determines how many of its own cycles will make up each full cycle giving wide control over the waveform shape. The timbre is always harmonic and often highly coloured.

The odd channel produces the modulated waveform and its amplitude controls work in the normal way.

The modulating waveform is a two-state sign-only version of the even channel's signal. Its resets the phase of the odd channel's waveform when either the modulating waveform is negative or it reaches the end of its cycle. The modulating waveform is usually null (preset waveform 13) so the reset occurs on cycles.

The tone of the modulated signal depends on:

1 the modulated waveform
2 the modulating waveform (usually null)
3 the frequency ratio

Its pitch is the pitch of the modulating channel.

With the modulated frequency higher than the modulating frequency, harmonics around that point are enhanced. At integral multiples, the pure modulated waveform is heard.

With the modulating frequency lower, the effect is to add harmonics that die away smoothly with increasing number.

Synchronisation can be very effectively used to create tone envelopes by using a pitch envelope on the modulated channel, for example:

```
1 EMOD ADSR 0 SUSTAIN % decay to zero
1 WMOD WZERO WGC % all zeros

SOUND 2 CHAN 1 WAVE 1 CHAN ON SYNC 1 PENV
SCAN SHIFT ON GATE % vary pitch and restrike
```

With a shallow pitch sweep, different base pitch values give many different tone change effects.

Frequency Modulation

Frequency modulation is a technique that makes complex tones by combining two signals. One signal (the carrier) has its frequency controlled by the other (the modulator). The waveform of the carrier is compressed and expanded, even within single cycles, creating a new waveform shape.

The odd channel's normal signal is used as the carrier, so the frequency modulation effect appears at its output with its amplitude controlled in the normal way.

The modulator is a two-state sign-only version of the even channel's signal, and is not affected by the normal amplitude controls. Its amplitude is set by FM and also multiplied by the frequency of the carrier so that the effective depth is constant with varying pitch. A depth of 192 produces + and - 100% modulation. More than 100% is allowed since the carrier frequency can go negative.

The tone of the modulated signal depends on:

1 the depth
2 the carrier waveform
3 the frequency ratio (pitch interval)
4 the modulating waveform

Assuming the depth is above zero, the most important of the these is the frequency ratio, equivalent to a pitch interval between the channels. Different intervals give different tones. Adding a large frequency offset to one oscillator gives a pitch-dependant interval, giving different tones for different pitches.

If the interval is simple, for example unison or an octave up or down, the sound will be harmonic. This means that the components will heard as a harmonic series, typical of simple vibrating sound sources like strings and air columns. Adding small frequency offset (by OFFSET) gives a wavering effect to the tone.

More discordant intervals produce non-harmonic sounds, like those of complex vibrating sources such as bells, gongs and chimes. The most dramatic effects appear when the modulator is at lower a pitch than the carrier. Sweeping either pitch produces widely varying and completely unnatural tones.

```
1 EMOD 1000 127 CYCLE    % slow up/down sweep

SOUND  1 CHAN 100 FM 400 PITCH 2 CHAN 1 PENV ON CHAN ON GATE
    % gate both channels
```

For most FM effects, a simple modulator waveform, for example a sine wave, is best. Note that the shapes of geometrically-created waveforms may not be 50% positive and 50% negative and will therefore give a net pitch shift when used on the modulator.

Multi-part Pieces

Players

One of AMPLE's most powerful features is concurrency - the ability to do more than one task at the same time. This allows it to play multi-part music in the same way as a group of human players, where the separately scored parts are played alongside each other to make the complete performance.

The computer's memory and processing power and the synthesiser's voices are divided among the players so that they work independently as if they had separate computers and synthesisers. Each one runs its own program, which in the case of a multi-part piece, is the score of one of the parts.

A special player looks after the sections of the system that cannot or should not be divided among the players. These include the keyboard, screen and other parts of the computer and you. This 'static' player always exists in the system, to execute commands and user words entered at the keyboard. When playing multi-part music, the static player does the job of the musical director, setting up the other, dynamic, players and starting them going. Once the piece has started, the static player returns to the keyboard to accept further commands while the piece is playing.

PLAYERS creates the specified number of dynamic players, discarding any already in existence, for example:

```
3 PLAYERS
```

Up to eight are allowed. The MEM command shows how much space is being used by players: each one consumes around 300 bytes.

The PLAY(...)PLAY structure is used to give a specified player something to do. Like all other structures, it can only be used inside words, for example:

```
"rep" ← SCORE REP( 0:CGdafGAB )REP →  % repeat tune

"play" ←
1 PLAYERS   % create one player
1 PLAY( SOUND rep )PLAY → % give it 'rep' to do

play                               % do it
```

Since the static player doesn't execute the PLAY contents itself, but just tells the numbered player to do so, it finishes and returns to the keyboard. Player 1 then plays 'rep' in the background.

The PLAY structure can include any sound or music-playing words that you would run in the static player.

Commands and Errors

You can now enter commands while 'rep' is playing, for example:

```
 600 TEMPO to speed it up
 ON FREEZE to halt it temporarily
 OFF FREEZE to restart it
 1000 FAST to run forward by 1000 units
 STOP  to stop and discard the player
```

You can use most commands without affecting the player, but players are discarded by command such as DELETE, LOAD, NEW etc.

which remove words that players could be using at the time. This is also the case when a word is redefined.

If you use a command that takes more than a few seconds, such as DFS *TYPE, the player will pause until the operation is finished and then run fast to make up for the lost time.

ESCAPE stops all players. Errors in the static player do not stop dynamic players, but an error in any dynamic player stops all of them. If an error occurs in a dynamic player, the player number is indicated in the error message, for example:

Player 1: Bad bar

This could pop up in the middle of some other operation in the static player, even while you are typing a command.

More players

To make two players play at the same time, we give each one a PLAY in turn, for example:

```
"play" ←
2 PLAYERS     % create 2 players
1 PLAY( SOUND 3 POS rep )PLAY  % player 1 does rep 2 PLAY( SOUND
-3 POS 50 DURATION rep )PLAY % player 2 has delay
GO →      % set players going
```

The 'rep' sequence is repeated by player 2 after a short delay, and the main and repeated sequences are at opposite sides of the stereo field.

Whenever two or more players are used together, a GO should be put immediately after the group of PLAYs to start all the players in synchronisation.

Each new player initially has one voice with two channels, and all channels are selected, just like the static player. VOICES assigns voices to the player it is used in, and these voices are completely separate from other player's voices. They are numbered separately for each player, in the same way as the channels of a voice.

Before eight players can be used, the static player's voices must be freed by 0 VOICES so that each of the dynamic players can be given one voice.

Multi-part Pieces

Here's are two examples to illustrate the complete form of the main word of a multi-part piece:

```
"play" ←
0 VOICES     % ensure all voices are free
1000 TEMPO     % set global tempo
2 PLAYERS     % create two players
1 PLAY( ins1 part1 )PLAY  % player 1's instrument and score
2 PLAY( ins2 part2 )PLAY  % player 2's instrument and score
GO →      % start both players together

"play" ← 0 VOICES 1000 TEMPO
4 PLAYERS     % create 4 players
1 PLAY( bassins basspart ↑ )PLAY % player 1's instrument & score
2 PLAY( leadins leadpart ↑ )PLAY % player 2's instrument & score
3 PLAY( 3 VOICES    % player 3 has 3 voices
 1 VOICE rtmins    % do instrument
 2 VOICE rtmins    % on all voices
 3 VOICE rtmins
 rtmpart ↑ )PLAY  % player 3's score
4 PLAY( 2 VOICES    % player 4 has 2 voices
 1 VOICE perc1ins  % voice 1's instrument
 2 VOICE perc2ins          % voice 2's instrument
 percpart ↑ )PLAY  % player 4's score
GO →      % start all players in sync
```

Programming

As well as sound and music words, and commands to manage user-defined words, AMPLE includes a large number of programming words, some of which are used routinely in the sound and music side. These words perform operations on numbers and strings, input and output, and control of program execution.

A full explanation of the programming side of AMPLE is beyond the scope of this manual, but the following notes plus the Reference Section will allow those with existing knowledge of programming write more advanced programs.

Numbers and Logical Values

Numbers are 16-bit 2's complement signed integers. The logical values ON and OFF are represented by the numbers -1 and 0 respectively.

Number operators are post-fix, working on the player-local number stack.

Numeric variables can be created with GVAR and PVAR.

Strings

A string is a sequence of up to 127 characters. String operators are post-fix, working on the global string stack.

The system uses the string stack for command input, so when a word is entered by the CLI, the top item is always the remainder of the input line, and on exit the top item will be interpreted as such. Hence most user words will leave the sring stack as they find it.

A sequence of characters delimited by " is accepted as a string literal and left on the string stack.

Examples: "hello"   ""

Compiled quoted strings are left on the top of the stack at execution time, but in direct mode the string is left as the second item, under the input line.

String operators can only be used inside words.

Control Structures

Control structures can only be used inside words i.e. cannot be used as commands.

The following control structures are available:

```
Conditional:  IF( ... )IF
    IF( ... )ELSE( ... )IF

Definite loop:  FOR( ... INDEX ... )FOR

Indefinite loop: REP( ... )UNTIL( ... )REP

Concurrent process: PLAY( ... )PLAY

Playing action: NOTE( ... )NOTE
    REST( ... )REST
    TIE( ... )TIE
```

11 Scoring from sheet music

This chapter gives some suggestions and guidelines for scoring pieces in AMPLE from sheet music. Much of it also applies to music written directly in AMPLE and will be of particular value if you are used to using conventional music notation.

Parts and Players

An important first step is to work out how many players will be required, and how many voices each will have. This decision will be based on the nature of the musical parts of the piece.

In the simplest case, the piece consists of a number of parallel monophonic parts. These will be be scored separately and performed by separate players, each with a single voice.

If there is an accompaniment part for a polyphonic instrument that plays well-defined chords, for example an acoustic guitar, this will be scored for one player using the note group facility to play the chords. If the chords are picked instead of strummed, it can be easier to set the voice with ';' to play notes on particular strings.

It is important to identify any separate musical line which should be scored separately, since this can get very complicated if entwined around the chords in th.i3.r67.w54.n
e same part. This applies particularly to polyphonic keyboard parts which are doing more than playing simple chords. It may be worth dividing the part into a number of parallel parts and scoring them separately, especially if there is much counterpoint.

Some parts may have more than one instrument; a drum kit part for example. This will be played by a single player having a voice for each instrument. The voice can be selected by ';', or since there will be a large number of voice changes, each instrument can be given its own voice-specific symbol, as explained later on.

Further players can be used for silent parts - those that control the music but do not play themselves. A good example of this is a conductor that controls the tempo to produce effects like ralletando and ritardando. Ties are used to mark time between the changes of tempo.

Silent parts can also be used to give control over certain parameters of the music from the keyboard but without tying-up the static player. The TAB-hold example given for IDLE in the Reference Section is an example of this - it runs in a spare player and freezes the timebase while the TAB key is down.

Note Values

Since the actual amount of time that each note length unit represents can be set over a wide range by TEMPO, you are free to choose the number of length units that will represent the smallest note length in the piece. The suggested values based on 48 for a crotchet allow divisions by two and three down to very short notes, but if the piece has, for example, a semiquaver as its shortest note, then this can be represented by a length of 1 and the crotchet will then be 4. It is a good idea to leave a margin of a few extra divisions at the short end, since it would otherwise be impossible to add shorter notes.

Bars

Bar lines are strongly recommended for music taken from printed form. Each line checks the length of the previous bar and produces and error if it is incorrect.

Bar lines are not used to synchronise parts. Once the players are started together by GO, they are held in alignment by the master timebase and there is no possibility of them getting out of step accidentally. Different bar lengths can be used in the different parts, and the length can be changed from bar to bar, since it is the note length unit that is controlled by the timebase.

If bar checking is required, the length of the bar must be specified with BAR, which takes the length in note length units. Checking can be disabled during testing by setting the length to 0.

To check for bad bars in a part without playing it through normally, use FAST to run through the part at maximum speed.

Signatures

Most music has the same time and key signatures for all parts, so to avoid duplicating these, a signature word can be defined and then used at the start of each part. This will contain:

1 SCORE to prepare for music words
2 the key signature

3 the bar length setting (representing the time signature)

Here's an example:

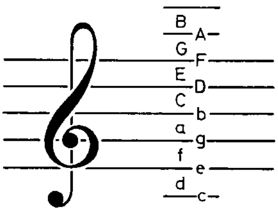
```
 % G major, 192 units to the bar
 "signature" ← SCORE  K( +F )K  192 BAR →

 % Used by each part like this:
 1 PLAY( instr1 signature page1 ... )PLAY
```

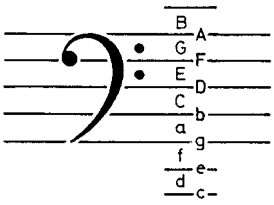
Clefs

Since AMPLE has no stave to fix the pitch of notes, there is no direct equivalent to a clef. However, the ':' word is usually used to fix the pitch of the first note after SCORE. 1: and -1: give a convenient range of starting notes for the G (treble) and F (bass) clefs respectively.

1: TREBLE CLEF



-1: BASS CLEF



Accidentals

In conventional notation, accidentals apply to all notes on the same line up to the next bar line. In AMPLE, they apply to the next note only, so an accidental must be repeated for repeated notes in the same bar.

It is easy to forget this when working through the music note-by-note, so the best approach is to first go through writing-in the held accidentals. You can also cross out any 'cautionary' accidentals at this stage.

Pages

In order to keep words down to a manageable size, each part should be written with a separate word for each page. The pages are then used in sequence in the PLAY structure, or strung together to make a complete part word. For example:

```
 % one part - 10's digit is part no., 1's digit is page no.
 1 PLAY( instr1 signature page11 page12 page13 ↑ )PLAY
 % or..
 "part1" ← page11 page12 page 13 →
```

For clarity, the pitch should be fixed with ':' at the start of each page.

The terminating rest (included at the end of each part to silence the last note) should be at the outermost level, just before the)PLAY. This makes clear that it is not part of the music.

The maximum convenient size for a word is about 25 lines - small enough to fit on the screen when listed.

Repeating Sections

Where the same section or phrase appears more than once in a score, it can be defined as a word and then named for each occurence, for example:

```
"song" ← intro verse chorus verse chorus chorus ↑ →
```

It is a good idea to fix the pitch at the start of each section word.

If a phrase repeats more than a few times in one go, a FOR loop can be used:

```
"drumpart" ← 15 FOR( pattern1 )FOR break
      15 FOR( pattern2 )FOR break →
```

Voices and Instruments

Most pieces make straightforward use of voices - each player has a certain number of voices which are set up with their instruments at the start of the piece and notes go to the voice set by ';' or implicitly by (...). More advanced techniques can be used to extend the use of the eight voices.

SHARE allows one player to use another's group of voices. Any number of players can share a single player's voices. The shared voices respond to sound words of all the sharing players, including the owner. This allows one player to use some voices when it needs them, with another using them for the rest of the time. It is important to get the players co-operating properly if the sharing is to be transparent. One important point is what each player does when it is not using the voices - it should play ties, which have no effect on sound, rather than rests, which gate the envelopes off.

Another technique uses voices which are shared by instruments. This works well for multi-instrument parts like drum kit parts, where ideally each instrument would have its own voice, but in fact they can share since rarely do all instruments play at once.

The simplest case uses a group of instruments that can be played as the same instrument at different pitches, and that don't need to play at the same time. Tom-toms are a good example. A single instrument definition is used to set up the sound, and different notes play the different 'instruments'.

Taking this further, any sound parameter, not just the pitch, can be set 'on the fly'. Even a complete instrument definition can be executed for each strike. The best way to do this is to define your own music symbols for the various instruments and use them in the score. Each one should set up the sound and strike the envelopes, either by ON GATE or preferably by a note so that the music voice and note length still apply. The most efficient method is to use rests to strike (redefining them with REST(...)REST) and use ties for quiet beats.

On-the-fly instrument definitions should be kept simple to avoid giving the player too much work. They need not use SOUND, setting up only those sound parameters that are needed, and leaving those that are common to all instruments on the voice. SOUND must still be used to set up the voice at the start of the part.

12 Glossary of terms

Accidental
a temporary modification of the pitch of a note, indicated by one or more sharp or flat signs before it.

ADSR
a simple form of envelope usually used for amplitude. It stands for ATTACK, DECAY, SUSTAIN, RELEASE - the four basic parameters needed to model the amplitude envelopes of most natural instruments.

Amplitude
a measure of the loudness of a signal at a particular instant.

Aperiodic (adjective)
see Non-periodic.

Attack
the build-up of amplitude from zero at the start of a note. The sound of a piano has an immediate attack, whereas that of a flute or organ is longer.

Bar
a division of musical time. Every bar has the same total of note lengths, and therefore, with constant tempo, each lasts for the same amount of time.

Bar line
the instruction that ends one bar and starts the next.

Duration
the amount of time between two events (in particular between one group of sound-changing commands and the next) or, as a result of this, the amount of time that an effect lasts for.

Channel
The basic sound-generating part of the music synthesiser. There are sixteen channels and each produces a single sound signal with its own pitch, tone, loudness and stereo position.

Chord
a group of notes that sound at the same time, usually starting and ending together. The notes are written together as a group but they play on different voices.

Chorusing
an effect used to make sound richer by playing similar waveforms together with slightly different frequencies. The frequency difference is greater than for phasing, and the separate signals are sometimes heard as a chorus of sounds.

Decay
in particular, the decrease in amplitude after the start of a note. The sound of a piano has a pronounced decay, whereas that of an organ has none.

Envelope
the shape of some aspect of a sound as it varies over time. The commonest types are the amplitude envelope and the pitch envelope.

Flat
an instruction that lowers the pitch of the next note by one semitone, or if in the key signature, does the same for all notes of a particular name.

Frequency
a scientific measure of how 'high' or 'low' a sound is: the rate of repetition of the sound's vibration pattern. Frequency is stated in Hertz (Hz); the number of cycles per second. Middle C has a frequency of 261.6 Hz.

Frequency modulation (FM)
rapid variation of the frequency of one signal by another signal, producing new timbres. FM sounds are vey complex (like bell and gong sounds) and often have ambiguous pitches.

Fundamental
the lowest sinewave component of a periodic waveform (the first harmonic). It sounds at the same pitch as the complete waveform, and is often the strongest harmonic (that is, has the greatest amplitude).

Gate
the signal that controls an envelope generator. Gate 'on' send the envelope to the start of its 'on' section, and gate 'off' sends it to the start of its 'off' section. In the case of most amplitude envelopes, the on sction lets the sound through and the off section turns it off. Notes send gate on signals, rests send gate off signals, and ties send no gate signals.

Harmonic
a single sinewave component of a periodic waveform. Any periodic waveform can be thought of as a series of sinewaves of different frequencies, its 'harmonics', added together. Their relative amplitudes determine the waveform's shape and therefore the timbre of the sound.

Instrument
the object that determines the sound used by a musical part. The instrument is responsible for aspects of the sound that are fixed from note-to-note, such as timbre, but not those that vary in the playing of the music, such as pitch.

Key
the particular set of pitches that a peice uses for the notes of the scale, described as the starting note of the scale (the 'key note') and the type of the scale, major or minor.

Key signature
the instruction to modify the pitches of the notes to suit the key of the music following. The normal pitches of the notes suits the key of C major, but since the interval between adjacent notes varies up the scale, the note pitches need to be modified for every other key to keep the same relative pitches at the new starting note.

Modulation
rapid variation of a signal's parameter by another signal, producing new timbres. In music, a modulation is a change of key in the middle of a peice of music.

Noise
a particular type of sound that has no identifiable pitch. The sounds of waterfalls, waves, hissing steam and cymbals are all noise-type sounds. Noise can be thought of as a mixture of an infinite number of different frequencies, with their relative amplitudes determining whether the noise is 'high' or 'low', rough or smooth etc. In practice, noise is produced by ring modulation using pseudo-random waveforms.

Non-periodic (adjective)
having no identifiable repeating period. Non-periodic waveforms are typical of complex vibrating objects like bells and gongs.

Note
a sound of a particular pitch and length; the fundamental unit of most music. The note names A to G refer to seven particular semitone pitches in any octave.

Overtone
another word for harmonic. It can also mean those partials that are above the perceived pitch of a non-periodic tone.

Partial
a single sinewave component of a waveform. If the frequencies of the components are integral multiples of a fundamental frequency, the waveform is periodic and the components are usually called harmonics. If no audible fudamental frequency can be identified, the waveform is non-periodic and the components are usually called partials.

Periodic (adjective)
repeats exactly after an identifiable interval of time. Periodic waveforms are produced by simple vibrating objects such as strings and air columns.

Phase
a position in the cycle of a waveform, stated in degrees (a complete cycle is 360 degrees). A 'phase difference' is the separation between related points in the waveforms of two signals playing together. Setting the phase of a waveform is starting it off from a particular point in its cycle.

Phasing
the sound produced by two similar waveforms with very slightly different frequencies. As the phase difference slowly changes, different harmonics cancel out, giving a jet-plane-type swooshing sound.

Pitch
the musical measure of how 'high' or 'low' a sound is. The pitch of a note is usually described as octave, measured from the octave range above middle C, and note name (C, D# etc), indicating the semitone within the octave.

Player
an object that plays a single scored part of music (or sequence of instructions), often alongside other players. The static player always exists, to handle instructions typed at the keyboard. Up to eight dynamic players are created specially for playing multi-part music.

Playing action
the sequence of steps that turns musical information supplied by a note, rest or tie into the appropriate sound - the interface between the score and the instrument, in fact. The action can be modified to create special playing effects such as staccato and accents, or replaced to change the interpretation of music events.

Release
that part of the sound of a note which apears after the note is released. Most natural instruments have an immediate release, but in the case of percussive instruments, the whole sound can be considered to be the release.

Rest
a period of silence in a musical part. Rests have lengths just like notes, and it is useful to think of them as notes which are 'off' and therefore have no pitch. Some instruments have sounds that carry on after the note has officially finished (see Release), so there may not in fact be silence while a rest plays.

Ring modulation (RM)
a rapid variation of the amplitude of one signal by another, named after the ring-shaped electronic circuit first used to produce it. The new timbres produced by the form of ring modulation used here include complex distorted timbres and pitchless noise-based sounds.

Semitone
a small unit of pitch; the smallest used in most music. The smallest difference between adjacent notes in a scale is one semitone, for example, between E and F. Two semitones equal one tone.

Scale
the group of different note pitches in one octave, often played up or down in order. The commonest types of scale are major and minor, which each have seven notes to the octave.

Sharp
an instruction that raises the pitch of the next note by one semitone, or if in the key signature, does the same for all notes of a particular name.

Sinewave
the simplest waveform. The sound of a sinewave is very pure and plain, with no brightness, colour or distortion. The on-the-hour pips of the Grenwich time signal and the TV end-of-transmission tone are both sinewaves.

Sustain
the holding-on of the sound while the note plays. It can also mean a continuing sound after the note has finished, as in the case of sustain stops on an electronic organ. In an ADSR envelope, the sustain level is the level to which the amplitude eventually settles when the note is held on.

Synchronisation
a form of modulation where the waveform of one signal is distorted by synchronising it to another, producing strongly-coloured (for example vowel-like) timbres.

Tempo
the speed of a peice of music. As well as the note lengths used in the music, the tempo depends on timebase period, which can be varied after the peice has been scored, even while it is playing.

Tie
an instruction that extends the previous note. The tie merely adds to the length of the previos note so it sounds as if it had been written with a longer length, but allowing it to extend over another instruction such as a bar line or, in AMPLE, a note played on another voice. Rests and ties can also be extended by ties.

Timebase
the internal reference which controls all durations in a peice of music, like a conductor's baton.

Time signature
an indication of the length of the bar for the music following. It has no effect on the sound of the music, but in AMPLE it is used to detect incomplete and extended bars.

Timbre
the 'tone' or 'quality' of a sound, as opposed to its pitch,

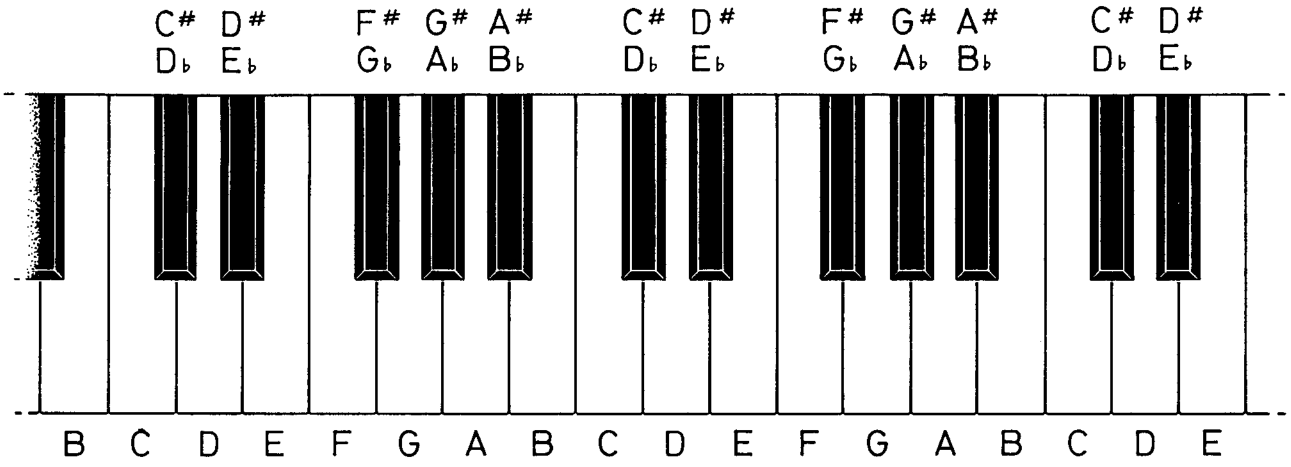
loudness, envelopes etc. Waveform and modulation determine the timbre of a synthesised sound.

Tremelo
a repeating variation in amplitude of a note as a feature of the instrument. It is characteristic of the sound of the flute. In natural instruments it is often accompanied by vibrato, and the two effects are often confused.

Vibrato
a repeating wavering imposed on the basic pitch of each note by the instrument. The rate is usually between two and six cycles each second. Vibrato is characteristic of electronic organ sounds.

Voice
an individual music-playing unit of the music synthesiser which can have its own instrument and its own notes to play. There are eight voices available. Each voice has a variable number of channels.

Waveform
the shape of a sound's vibration pattern. The waveform determines the timbre of a sound. A periodic sound has an easily-identifiable waveform since it repeats on each cycle, but a non-periodic sound has no identifiable repeating cycle and therefore no fixed waveform.



KEYBOARD

13 Errors

This chapter describes AMPLE's error system.

Error reporting and effects

When a fault is found in the input line or in an AMPLE word in the input line, an error message is printed, for example:

Mistake

The rest of the line is ignored.

When an error occurs in a user word executed in the static player (that is, directly or indirectly from the keyboard but not in a dynamic player), the message is printed with the name of the user word, for example:

No such voice in tune

Execution of the static player stops and control returns to the keyboard. The note context is set to normal.

When an error occurs in a dynamic player, the player number is also given, for example:

Player 1: Bad bar in page1

All dynamic players and sounds are stopped and the timebase is unfrozen. The static player's is unshared, that is, set to use its own voices, and its note context is set to normal.

ESCAPE presses are treated like errors in players, but no player number is given.

If an error occurs in the definition of a word, that is, between ← and →, a ! is printed below the faulty characters.

REPORT shows the location of the last error that occured in a user word.

Error Messages

The error messages are listed here in alphabetical order:

***<message>**
A serious fault has arisen in the system, probably as a result of memory being corrupted.

If this occurs, you should press CTRL BREAK and reload the language. You can save the program first, but in extreme cases this may also have been corrupted and will be rejected by LOAD.

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

Bad bar
The total length of a bar did not match the bar length set by the time signature.

Bad chord
A chord group was found inside a chord or key signature.

Bad command
A word that is only allowed inside word definitions was enetered directly as a command. Control structure words (PLAY(IF(etc.) are of this type.

Bad hex
The first character after & was not a valid hex digit.

Bad key sig
a key signature was found inside a chord or key signature, or the end of a key signature was found without a beginning.

Bad mode
There was not enough free memory for the mode requested. MODE does a COMPACT before checking the free memory.

Bad name
There was a fault in the name given for a new word. It was probably too long (longer than 15 characters).

Bad player number
A number outside the range 0 to 8 was used as a player number. SHARE can give this error.

Bad program
The loaded file was not an AMPLE program produced by SAVE.

This error leaves a null program, as if you had used NEW.

Bad section
An invalid number of segments was specified for a section in ESECT.

Bad string
There was no closing " in the string.

Bad structure
The end part of a structure did not match the last unmatched

beginning part, for example

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

Bad voice
An attempt was made to assign channels to voice 0, the dummy voice.

Command only
A word that is only allowed to be used directly appeared inside a word definition. These include commands that destroy words, like NEW and DELETE.

Division by zero
An attempt was made to divide by zero.

Escape
The ESCAPE key was pressed. ESCAPE stops all players and sounds.

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.

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
```

You may have attempted to include a quote in a string by repeating it, which is not allowed, for example

```
"""hello""" $OUT    ("" and "hello" remain)
```

In player
A word whose use by a player is not allowed was used by a player, for example

```
1 PLAY( GO )PLAY
```

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

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

```
PRINT         (should be ON PRINT -
              ON leaves a number)
```

In complicated programs that use the stack for temporary storage, this often results from a programming fault. If there was 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 such channel
An attempt was made to use a channel which the current voice didn't own.

This can be the result of a simple mistake, such as

```
1 VOICE 4 CHAN   (CHANS was intended)
```

or failing to select the voice:

```
3 VOICES SOUND
```

No such envelope
An envelope number outside the range 1 to 10 (and not one of SIMPLEP or SIMPLEA) was used.

No such harmonic
A harmonic number outside the range 1 to 16 was used.

No such point
A waveform point number outside the range 1 to 128 was used.

No such segment
The specified segment did not exist in the current envelope.

No such voice
An attempt was made to use a voice which the player didn't own.

Players have one voice as standard, so, for example, you cannot play chords without first assigning (and setting up with SOUND) some more voices.

No such waveform
A waveform number outside the range 1 to 13 (and not SIMPLEW) was used.

No such word
The word did not exist. Commands such as EDIT or DELETE produce this error.

No room
There was not enough free memory for the operation. This error can be produced in a word definition, by EDIT, and by PLAYERS.

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 peice.

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 STOP.

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.

String too long
The maximum total length of strings on the string stack was exceeded. It can hold 128 characters.

Note that since the input line is held on the stack, a word which uses long strings may work normally, but give this error if executed from a long input line.

Too many channels
All sixteen channels were already in use when an attempt was made to assign some to a voice.

Channels can only be assigned in pairs - if you ask for an odd number you will get one more than this, so the following will produce an error:

```
1 VOICE 5 CHANS 2 VOICE 5 CHANS 3 VOICE 5 CHANS
```

Too many levels

The capacity of the player's return stack was exceeded.

This happens in the following cases:

1a too-deep nesting of words was used
2a too-deep nesting of FOR loops was used
3a player was given too many PLAY structures to do in advance

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

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

Too many segments
A total of greater than 10 segments was asked for by ESECT.

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

Too many voices
All eight voices were in use when an attempt was made to assign some to a player.

Remember that the static player uses one voice, so this must be freed with 0 VOICES before you can use all eight players.

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

Word deleted
A word used in a word definition had since been deleted.

Reference section

This chapter contains a concise description of every AMPLE word.

Each word has its own entry. The entries are arranged by name in a lexicographic (dictionary-type) order based on the following order of characters:

! " # $ % & ' ( ) * + , - . / : ; < = > ? ← \ → ↑ A to Z

In names that include one or more letters, leading non-letters have the significance of trailing characters, that is, they only affect the ordering of otherwise-identical names. This means that, for example, &VAL appears after VAL even though & appears before VAL.

The general form of an entry is as follows:

```
word name   function                                    status
            arguments -> results
```

description

example(s)

In almost all cases, the word name appears exactly as you would type it in, but in some cases a description of the name, in angle brackets, is given instead. For example, <carriage return> means the carriage return character. Where a name includes spaces, they appear as ~sp .

A short indication 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 is a command - it cannot be used inside words.
←→ only	The word can only be used between ← and →, that is, only inside words and not typed in directly as a command.

Where there is no status, the word can be used both as a command and inside words.

For those words that take arguments and/or return results, these are indicated below the function with arguments on the left of the arrow and results on the right. A dot on either side stands for no items.

Note that the order of numbers/flags and the order of strings are important on both sides, but strings and numbers/flags are supplied and returned separately so the order of two items of different type is not important.

<carriage return>   do nothing

The word with a carriage return as its name does nothing when executed.

Its function is to represent line ends inside definitions.

~sp   do nothing

The word with a single space as its name does nothing when executed.

Its function is to represent spaces between words inside definitions.

~sp~sp~sp~sp~sp~sp   do nothing

The word with six spaces as its name does nothing when executed.

Its function is to represent groups of spaces between words inside definitions more compactly than the single space word.

```
"      start literal string
       . -> string                             in a word
       inputstring1 -> string inputstring2     in direct mode
```

The characters up to the next " on the same line are accepted as a string. When executed inside a word the string is left as the top item on the string stack, whereas in direct mode it is left under the top item (the input line).

```
Examples:       "prog" LOAD              % "prog" is left for LOAD
                ← .. "Value:" .. →       % "Value:" is left on top
```

```
#!     store number at address
       number address -> .
```

The number is stored at the address on top of it.

#! is used for assigning values to variables, and in most cases it appears immediately after the variable name. It will operate on any address and should therefore be used with care to avoid corrupting memory.

```
Example:        0 total #!        % set variable 'total' to zero
```

#*   multiply two numbers

number1 number2 -> productnumber

The top two numbers are multiplied together, leaving the result.

Example: -2 3 #* goes to -6

#+ add two numbers
number1 number2 -> sumnumber

The top two numbers are added together, leaving the result.

Example: 2 3 #+ goes to 5

#+! add number to number at address
number address -> .

The number is added to the number at the address on top of it.

#+! is used for adding numbers to variables and in most cases it appears immediately after the variable name. It will operate on any address and should therefore be used with care to avoid corrupting memory.

```
Example:        1 total #+!      % add one to total
                                 % (previously-defined variable)
```

#- subtract number from previous number
number1 number2 -> differencenumber

The top number is subtracted from the one below, leaving the result.

Example: 3 2 #- goes to 1

#/ divide previous number by number
number1 number2 -> quotient remainder

The top number is divided into the number below, leaving the quotient with the remainder on top.

```
Examples:       10 3 #/          goes to   3 1    % full result
                10 3 #/ #2       goes to   3      % quotient only
                10 3 #/ #12 #2   goes to   1      % remainder only
```

#11 duplicate number
number -> number number

A copy is made of the top number.

```
Examples:        4 #11    goes to    4 4
                 #11 0 #= IF( ... )IF     % non-destructive test
```

#12 swap two numbers
number2 number1 -> number1 number2

The top two numbers are exchanged.

```
Examples:        8 5 #12    goes to    5 8
                 "MOD" ← #/ #12 #2 →      % num1 num2 -> remainder
```

#2 discard number
number -> .

The top number is discarded.

```
Example:         2 1 #2    goes to    2
```

#212 duplicate previous number
number2 number1 -> number2 number1 number2

The number under the top number is copied to the top.

```
Example:         3 2 1 #212    goes to    3 2 1 2
```

#2121 duplicate number and previous number
number2 number1 -> number2 number1 number2 number1

Copies are made of the top two numbers.

```
Examples:        4 7 #2121    goes to    4 7 4 7
                 #2121 #= IF( ... )IF     % non-destructive test
```

#213 rotate positions of top three numbers
number3 number2 number1 -> number2 number1 number3

The number two down from the top is moved to the top.

```
Example:         3 2 1 #213    goes to    2 1 3
```

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

A flag is left which is ON if the top number was less than the one below it, and OFF otherwise.

```
Examples:        4 0 #<    goes to    OFF
                 4 6 #<    goes to    ON
```

#= **test numbers are equal**
number1 number2 -> flag

A flag is left which is ON if the top two numbers were equal, and OFF otherwise.

```
Examples:        4 4 #=    goes to    ON
                 4 5 #=    goes to    OFF
```

#> **test previous number is greater than number**
number1 number2 -> flag

A flag is left which is ON if the number was greater than the one on top of it, and OFF otherwise.

```
Examples:        -2 0 #>    goes to    OFF
                  3 2 #>    goes to    ON
```

#? **fetch number from address**
address -> number

The address is replaced by the two-byte number at the address.

#? is mainly used for extracting values from variables.

```
Example:         total #? NOUT    % print value of variable 'total'
                                  % (previously-defined variable)
```

$+ **add string to left end of previous string** **↔ only**
right-string left-string -> string

The top string is added to the left end of the string underneath.

Examples:

```
        " there" "hello" $+    goes to    "hello there"
```

$- **split string after numbered character** **↔ only**
string number -> right-string left-string

The string is split after the given character position, leaving the left part with the remaining right part underneath.

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.

Examples:

```
"hello" 2 $-    goes to    "llo" "he"

% extract middle substring
% string lennumber startnumber -> substring
"$mid" ← $- $2 $- $12 $2 →
"hello" 2 1 $mid    goes to    "el"
```

$12   swap two strings   ↔ only

string1 string2 -> string2 string1

The top two strings are exchanged.

Example: "hello" "there" $12   goes to   "there" "hello"

$2   discard string   ↔ only

string -> .

The top string is discarded.

Example: "hello" "there" $2   goes to   "hello"

%   introduce comment

% causes the rest of the line to be treated as a comment, i.e. ignored.

Example: ON CHAN 128 AMP % all channels sounding

&   indicate hex number

& precedes a hex number, distinguishing it from a decimal number or sequence of note names.

& searches up to the next non-digit, so there must be no spaces between digits or between & and the first digit.

Examples:	&FF	is equivalent to	255
	&8000	is equivalent to	-32768

'   indicate AMPLE word

' is used before a word to make sure that it is understood as an AMPLE word and not as a user word of the same name.

Its function is to override user words which, in order to replace AMPLE words, have deliberately been given the same names.

One use is in the definition of a user word which both uses and replaces an AMPLE word or word sequence. The ' is put before the AMPLE word name to prevent a re-definition of the user word from using itself instead of the original AMPLE word. This would otherwise happen if the word was edited, or was re-created from a file made with WRITE.

```
Examples:       % replace | with counting version
                "|" ← '|                 % normal bar-line...
                1 barcount #+! →         % plus count using 'barcount'

                % define common note lengths as
                % single words to save memory space
                "24," ← '24, →
                "48," ← '48, →
```

(start parallel note group

(and) enclose a parallel note group, in which music values are local and the music voice advances after each music event (note, rest or tie). The events within the group play in parallel with the previous normal event outside the group.

Parallel note groups are used for playing chords and other polyphonic effects.

(keeps copies of the effective last note, note length and music voice, and sets the note length to zero.

With the note length at zero, the notes inside the group start simultaneously on separate voices, that is, they play with the main note (the previous note outside the group) as an unbroken chord. The first note plays on the voice one above the music voice that was in force on entry to the group, and successive notes play on successive voices. The notes of the group play for the length of the main note.

The music voice and note length can be set as normal within a group. Each event contributes its length to all voices and therefore adds to the total length of the group and the main note. When setting the voice number with ';', remember that the music voice is incremented before each event.

A parallel note group cannot contain another group or a key signature.

Examples:

```
        isolated chord             C(EG-B) ↑(↑↑↑)
        chord sequence             C(EG) /(/a) g(BD) f(AC)
                                or C(EG) (2;a) g(BD) f(AC)
        synchronised asides        ... (POW) ... (ZAP) ...
```

)   end parallel note group

(and) enclose a parallel note group, in which music values are local and the music voice advances after each note. The notes, rests and ties within the group play in parallel with the previous normal note.

) plays an @ to complete the last note of the group, and restores the effective last note, note length and music voice to their original values.

See (for more information.

***   send command to operating system   Command**

The rest of the line is sent to the operating system as a command.

The following types of command are not allowed:

1 Language entry commands, for example *BASIC
2 Memory-corrupting commands, for example Acorn DFS *COPY

```
Examples:     *CAT
              *FX12,4
```

+   sharpen next note

The pitch of the next note is raised by one semitone, overriding the key signature.

The total modification of a note can be up to plus and minus four semitones.

```
Examples:        +F        % F sharp
                 ++C       % C double-sharp
```

,   set note length
number -> .

',' sets the length of notes, rests, ties and note groups. The number is in timebase units and has a range of 0 to 32767.

The note length chosen to represent a particular note value, such

as a crotchet, is entirely arbitrary as far as AMPLE is concerned, and only the the ratio of note lengths is important to the music. The actual duration of a particular note length is detemined by TEMPO.

Suggested lengths for the note values are as follows:

hemidemisemiquaver	3	demisemiquaver	6
semiquaver	12	quaver	24
crotchet	48	minim	96
semibreve	192	breve	384

The note lengths of modified values such as dotted and triplet are found by multiplying the normal length by the appropriate factor:

dotted note	x 1.5
triplet note	x 2/3

The note length is passed to all playing actions as the top number, and it is usually used by DURATION.

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

- **flatten next note or indicate negative number**

- has two functions.

If the next character is a decimal digit, it accepts the characters up to the next non-digit as a negative decimal number.

If the next character is not a digit, it flattens the next note The pitch of the next note is lowered by one semitone, overriding the key signature.

The total modification of a note can be up to plus and minus four semitones.

```
Examples:    -200         % minus 200
             -B           % B flat
             --a          % A double-flat
```

. **put line in text buffer** **Command**

number -> .

The text after the '.' is put on the numbered line in the text buffer.

If there is a line with the same number in the buffer, it is replaced by the new one. If there are no characters after the '.', the existing line is deleted.

If there was no line with the same number, the new line is inserted in numerical order.

```
Examples:    20.play      % add line 20
```

```
        100.        % delete line 100
```

/   extend currently-playing notes

/ represents a tie and extends the notes playing on all the player's voices by the current note length.

It is used to increase the length of notes, tie notes over bar lines and other words, and to pass over voices in parallel note groups.

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

If the tie is in a parallel note group, it plays as described but the music voice is incremented by one afterwards. The note length is set to 0 on entry to a group, so unless the length is changed, the tie serves only to pass over a voice.

/ is a dummy note which has length but no pitch or gate effect, and therefore affects all voices. It calls the TIE playing action with the note length:

```
        . -> notelength          % as seen by the TIE action
```

Examples:

```
        48,f ! /             % two crotchets tied across bar line
        C(EA) /(/G) a(DG) % passing note in chord sequence

        ! D/// /D/D !        % 'rests' with percussion instrument
```

:   set music pitch
        number -> .

':' moves to a particular position in the pitch range, setting the effective last note to C in the octave indicated by the number. The next note plays immediately above (or at) this C if it is upper case and immediately below if lower case.

Middle C is numbered zero, lower Cs are negative numbers and higher Cs are positive numbers.

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
```

;   set music voice

';' selects the numbered voice as the one that notes and rests (outside parallel note groups) will play on.

The music voice is the voice that the pitch and gate parameters of a note are sent to, but the note's duration is not voice-specific and controls the whole player. This means that each note, rest or tie contributes its length to all the player's voices, so they stay in step. Since ties have no pitch or gate, they have the same effect whatever the music voice.

A note played on one voice continues until a new note or rest is played on the same voice, though other voices may have been selected and notes played on them in the meanwhile.

With the normal playing actions, the music voice set by ';' corresponds to the sound voice set by VOICE. The voice must have been assigned to the player by VOICES and set up by SOUND.

The music voice is passed to the NOTE and REST playing actions as the second number from the top, and is usually used by VOICE.

```
Example:    1;                           % use voice 1 - default
            16, 1;cDEF G 2;fGA B 3;aBCD % chord build-up effect
```

<   move down one octave

< lowers the effective last note by one octave so that the following notes play an octave lower than they would otherwise have done.

```
Examples:   DEc<cG↑      % second C is an octave below first C
```

=   naturalise next note

The next note plays at its unmodified pitch, that is, without the effect of the key signature.

```
Example:    =b           % B natural
            =F           % F natural
```

>   up octave

> raises the effective last note by one octave so that the following notes play an octave higher than they would otherwise have done.

```
Example:    CD>Ecba↑     % E is an octave and a semitone above D
```

???   give 'Word deleted' error   ↔ only

References to deleted words appear as ??? in the text of words.

??? gives an error on any attempt to use it.

Example: "main" ← part1 part2 ??? part4 → % part3 was deleted

@ finish note

@ finishes the previous musical event (note, rest or tie) explicitly.

Its function is to align changes of sound that occur between music events with the beginning of the next event, and it is used before the sound word, instrument word, etc.

Only the first sound word after a musical event needs to be preceded by @.

@ effectively plays a zero-length tie (using the TIE action).

Example: C @ ins D   % change instrument for D

← start word definition **Command**
string -> .

← defines a word of the given name to perform the sequence of existing words up to →.

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

If a user word of the same name already exists, it is replaced by the new definition such that all references to it in other words and text will use the new definition. Dynamic players and sounds are stopped, and the static player's actions are reset.

If there is an AMPLE word of the same name, it is not replaced but all future definitions (and re-definitions) will use the user word instead of the AMPLE word.

Any existing user or AMPLE word, other than AMPLE commands, can be used in the definition of a new word.

Between ← and →, the normal prompt is replaced by ←% .

Example: "cube" ← #11 #11 #* #* → % cube number

→ end word definition **↔ only**

← and → define a new word to perform the enclosed sequence of existing words.

→ terminates the definition, checking that there are no incomplete control structures, and inserts it on the list of user words.

See → for more information.

↑ play rest

↑ plays a rest of the current note length on the current music voice.

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

If the rest is in a parallel note group, it plays as normal but the music voice is incremented by one afterwards. The note length is set to 0 on entry to a group, so unless the length is changed, the rest starts simultaneously with notes on other voices in the group.

Rests are also used at the end of pieces. Since each note, rest, tie or note group plays until the next one starts, the last notes would play indefinitely of not followed by rests.

↑ calls the REST playing action, supplying note length and music voice:

```
        . -> musicvoice notelength        % as seen by REST
```

```
Examples:    C↑C↑C↑C↑            % isolated notes

             F(ACE) ↑(↑↑↑)       % isolated chord

             192,D ! ↑           % end of piece
```

! end bar

! represents a bar line. The function of bar lines is to detect missing and extra notes and they normally have no effect on the music. Their use is entirely optional.

Each bar line checks the total of the note lengths in the previous bar. If this is different from the bar length set by the last BAR, the 'Bad bar' error appears.

If the bar length is set to zero, checking is disabled - note lengths are still totalled but not checked.

SCORE sets the bar length to zero, so BAR must be used if bar checking is required.

SCORE sets the the total of note lengths in the bar to 0.

```
Examples:    SCORE 192 BAR  48,C E F c |        % ok
             SCORE 192 BAR  48,C E D |          % gives 'Bad bar'
             SCORE   0 BAR  48,C E D |          % not faulted
```

A play ascending A

'A' plays the note A above the previous note.

A note can appear in one of three contexts:

1 Normal
2 Parallel note group, (...)
3 Key signature, K(...)K

In normal and group contexts, each note word plays a note of the current note length on the current music voice. Its pitch is indicated by the letter, which determines the note of the scale, and its case which controls the octave relative to the last note. Lower-case notes play below the previous note and upper-case above, though a repeated note name always plays at the same pitch. The effective last note can be changed explicitly by ':', < and >.

The unmodified note name pitches are as follows:

C	0
D	32
E	64
F	80
G	112
A	144
B	176

If no +, - or = is used on the note, it will receive the modification set for all notes of its name by the last key signature. If +, - or = is used on the note, the key signature has no effect and each + and - modifies the pitch of the note by plus and minus one semitone respectively. = serves only to cancel the key signature modification for a note.

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

If the note is in a parallel note group, it plays as described but the music voice is incremented by one afterwards. The note length is set to 0 on entry to a group, so unless the length is changed, the note starts simultaneously with notes on other voices in the group.

If the note appears in a key signature, then it does not play, but stores any modification applied to it with + and - for application to future normal and group words of the same letter.

In normal and group contexts, the note calls the NOTE playing action:

```
      . -> notepitch musicvoice notelength    % as seen by NOTE
```

```
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'
```

a   play descending A

'a' plays the note A below the last note.

See A for more information.

ADSR   create ASDR envelope

ADSR makes the current envelope into a basic ADSR shape for use as an amplitude envelope. Its four basic parameters can then be changed by ATTACK, DECAY, SUSTAIN and RELEASE.

When the gate goes on, the envelope enters the attack segment and the value starts climbing towards the peak of 127. When it reaches the peak, it enters the decay segment and falls until it reaches the sustain level, where it stays until the next gate. When the gate goes off, the envelope enters the release segment and the value starts falling towards zero, where it stays until the next gate on.

The initial parameters of the ADSR envelope are:

```
  1 ATTACK       % immediate attack
100 DECAY        % medium decay
115 SUSTAIN      % slightly-lowered sustain level
 10 RELEASE      % soft release
```

Envelopes created by ADSR are in standard format, so the envelope

segment words EGRAD and ELEV can be used on them in the normal way. Its three sections have one segment each, with the repeat section performing both the decay and sustain parts of the ADSR.

Created by envelope segment words, the basic ADSR envelope looks like this:

```
    1 1 1 ESECT                    % setup sections
```

```
     127    1  1 EGRAD   127  1 ELEV    % On: attack to 127
    -127  100  2 EGRAD   115  2 ELEV    % Rep: decay to sustain level
    -127   10  3 EGRAD   0    3 ELEV    % Off: release to 0
```

There is no distinction between envelopes for pitch and amplitude, so ADSR envelopes can also be used for pitch.

```
Examples:    1 EMOD           % select envelope 1
             ADSR             % setup ADSR shape
             20 ATTACK        % set slower attack

             % ADSR for use as pitch envelope
             1 EMOD           % select envelope 1
             ADSR             % setup ADSR shape
             60 1 ELEV        % lower peak to 60
```

AGATE   set amplitude envelope state
        flag -> .

AGATE controls the current channel's amplitude envelope by setting it to the start of the On or Off section. It takes a flag that determines the sense of the gate:

```
ON AGATE     set gate on: move to start of On section
OFF AGATE    set gate off: move to start of Off section
```

Normal amplitude envelopes turn the sound on in the On section and off in the Off section, so the on and off gates correspond to notes and rests.

AGATE is used when independent control of the amplitude envelope is needed.

AENV   select amplitude envelope
        number -> .

AENV selects the numbered envelope for use by the current channel as its amplitude envelope.

The amplitude envelope of a channel controls the amplitude along with the AMP setting, varying the amplitude within each note.

The number is an envelope number in the range 1 to 10, or an envelope identifier left by SIMPLEA (or SIMPLEP).

Newly selected envelopes are at zero, giving silence, until they receive a gate.

SOUND performs SIMPLEA AENV.

```
Examples:    3 AENV           % select envelope 3
             SIMPLEA AENV     % restore default envelope
```

```
        organA #? AENV  % select envelope from variable
```

ALIGN ensure text cursor is at start of 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, it is moved to the start of the next line.

```
Example:    ALIGN "Enter pos:" $OUT
                          % inside word, put prompt at line start
```

AMP   set amplitude
number -> .

AMP controls the overall amplitude (volume) for the channel, along with the value of the channel's amplitude envelope.

The range of amplitude is 0 (off) to 128 (maximum). Control is logarithmic, so that the range is wide and the volume goes down rapidly as the number is decreased from 128. The wide range is used by the envelope, and the useful span for setting a channel's volume is from 100 upwards.

SOUND sets AMP to 128 on channel 1, and to zero for all other channels.

```
Example:    2 CHAN 120 AMP    % quiet channel 2
```

AMPLE restart language **Command**

AMPLE restarts the language from scratch.

The program and text are discarded.

APPEND   add text of word to buffer **Command**
string -> .

The text of the named user word is added to the text already in the buffer. The new lines are numbered 10 apart, starting at the last line number of the existing text plus 10.

```
Example:    "secondbit" APPEND
```

AND   AND bits of number with bits of previous number
number1 number2 -> number3

Each bit of the result is the logical AND of the corresponding bits of the two numbers. AND acts as a flag operator if both the numbers are flags (ON or OFF).

Examples:

```
&1234 &FF AND     goes to    &34

#11 0 #> #12 5 #< AND     % number -> flag
                          % test number in range 1 to 4
```

ASC convert character to number in ASCII **↔ only**
string -> number

The first character of the string is converted to its corresponding ASCII code. If the string was null, -1 is left.

```
Example:     "A" ASC       goes to    65
```

ATTACK set attack time of ADSR envelope
number -> .

ATTACK sets the attack time of the current envelope, which will normally have been set up by ADSR.

The attack is the initial build-up of the amplitude at the start of a note (when the envelope gate turns on).

ATTACK sets the time, in centisecond units, that it takes to go from zero to maximum (127 units). The range is 1 to 65535.

Note that the attack starts from whatever level the envelope is at when the gate goes on.

```
Example:     1 EMOD         % select envelope 1 for modification
             ADSR           % setup basic ADSR
             50 ATTACK      % set attack time of 50cs (0.5s)
```

B play ascending B

'B' plays the note B above the last note.

See A for more information.

b play descending B

'b' plays the note B below the last note.

See A for more information.

#B! store low byte of number at address
number address -> .

The low byte of the number is stored at the address on top of it.

In most cases #B! appears immediately after a variable name. It will operate on any address and should therefore be used with care to avoid corrupting memory.

Note that two-byte values are stored with the low byte at the address and the high byte at the address plus 1.

Example: 0 value 1 #+ #B! % store 0 in high byte of 'value'

#B12 swap high and low bytes of number
number1 -> number2

The high and low bytes of the number are exchanged.

Example: &1234 #B12 goes to &3412

#B? fetch byte from address
address -> number

The address is replaced by a number with the byte at the address as the low byte, and zero as the high byte.

Note that two-byte values are stored with the low byte at the address and the high byte at the address plus 1.

Example: value 1 #+ #B? % get high byte of 'value'

BAR set bar length

BAR sets the bar length - the required length of each bar in note length units - for checking by bar lines.

With the bar length set to 0, the checking action of bar lines is disabled.

SCORE performs a 0 BAR, so the bar length must be set for the bar lines to perform checking.

Examples:

```
192 BAR        % 192 units per bar for
4 64 #* BAR    % 4/4 time (4 crotchets per bar)

120 BAR        % 120 units per bar for
5 24 #* BAR    % 5/8 time (5 quavers per bar)
```

C play ascending C

'C' plays the note C above the last note.

See A for more information.

c play descending C

'c' plays the note C below the last note.

See A for more information.

CHAN select specified channel(s) of voice
number -> .

The specified channel or channels of the current voice are made the current channel(s), that is, the target of future channel-specific sound words. The channels of a voice are numbered from 1 upwards.

Only channels that the voice owns (those that have been assigned to it) can be selected.

If the argument is the number of a channel then that particular channel alone is selected. If the argument is a channel group identifier, then all the channels in the group are selected. The options are:

number CHAN	specified channel
ON CHAN	all channels
OFF CHAN	no channels
ODD CHAN	all odd channels
EVEN CHAN	all even channels
number PAIR CHAN	specified channel and other channel of pair

The initial state of the channel selection after a voice is selected is ON CHAN (all channels selected).

```
Examples:    ODD CHAN  ON RM          % ring mod for all

             "detune" €
             4 FOR( INDEX CHAN        % detune 4 channels
                  INDEX 200 #* OFFSET
               )FOR →
```

CHANS assign channels to voice
number -> .

CHANS assigns the specified number of channels to the current voice in place of any it previously had. All the channels are selected for control.

The number of channels asked for must be even, that is, channels can only be assigned in pairs. A pair consists of an odd-numbered channel and the even-numbered channel above it.

Voices initially have two channels each after being created by VOICES. Voice 0 cannot have channels assigned to it.

There is a total of 16 channels available.

```
Example:      16 CHANS     % put all channels on this voice
```

$CHR   convert number to character in ASCII   ↔ only
number -> string

The number is converted to its corresponding ASCII one-character string. If the number is negative, a null string is produced.

```
Examples:    65 $CHR        goes to    "A"
             -1 $CHR        goes to    ""
```

CLEAR   clear text buffer

The text buffer is cleared, preparing it for typing in new text.

CLEAR should be used with care as its effect is irreversible.

CODE   call machine-code routine
YX CA address -> YX PA

The machine-code routine at the address is called. As well as the address, CODE 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

```
Example:      1 15 &FFF4 CODE #2 #2 % *FX15,1 flush input buffer
```

COMPACT   compact unused memory   Command

COMPACT arranges unused memory space into one contiguous area, making it fully available for use. All dynamic players and sounds are stopped, and the static player's actions are reset.

The space freed by deleting words and lines of text can be left in isolated peices which are too small for re-use. When this happens, a 'No room' error can arise when there is enough space in total. COMPACT should be used and the operation that gave the error repeated.

CYCLE set depth and period of cycle

periodnumber depthnumber -> .

CYCLE makes the current envelope in to a repeating up/down shape for use as a pitch envelope.

CYCLE is used to create repeating pitch effects such as vibrato and siren-type pitch sweeps. Each cycle consists of an upward slope to a positive peak and an equal downwards slope to a negative peak. When the envelope gate goes on, the value is set to zero and then starts cycling with the upward slope. It continues until the next gate on, ignoring gate offs.

The depth argument (the top number) is the value of the peaks measured from zero. Its range is -127 to 127. When the envelope is used as a normal pitch envelope, these are 16th semitone pitch units. EBIG can be used in the normal way to increase the range of the pitch sweep.

Negative depths cause the 'positive' peak to be below the 'negative' peak, so the cycle starts on a down slope.

The period argument (the previous number) is the period of the cycle in centisecond units. Its range is 2 to 32767.

Envelopes created by CYCLE are in the standard format so that the envelope segment words EGRAD and ELEV can be used on them in the normal way.

The CYCLE envelope created by

```
        200 32 CYCLE
```

looks like this when created by envelope segment words:

```
          1    2 0 ESECT
        127    1 1 EGRAD    0 1 ELEV    % set to zero
         64  100 2 EGRAD   32 2 ELEV    % up over half period
        -64  100 3 EGRAD  -32 3 ELEV    % down over half period
```

Examples:

```
% fast vibrato
2 EMOD                 % select envelope 2 for modification
15 3 CYCLE             % 15cs period (6Hz) and +_3 depth

% siren pitch sweep
2 EMOD                 % select envelope 2 for modification
```

```
500 96 CYCLE        % 5s period and one octave total depth

% extreme sweep
2 EMOD              % select envelope 2 for modification
50 127 CYCLE        % 0.5s period, maximum depth
ON EBIG             % switch to x4 depth
```

D play ascending D

'D' plays the note D above the last note.

See A for more information.

d play descending D

'd' plays the note D below the last note.

See A for more information.

DECAY set decay time of ADSR envelope
number -> .

DECAY sets the decay time for the current envelope, which will normally have been set up by ADSR.

The decay is the fall of the amplitude to the sustain level after it has reached its maximum at the start of a note.

DECAY sets the time, in centisecond units, that it takes to go from the peak (127 units) to zero. Its range is 1 to 32767.

Note that the decay stops when the sustain level is reached.

```
Example:    1 EMOD        % select envelope 1 for modification
            ADSR          % set up basic ADSR
            200 DECAY     % set 2s for 127 to 0
            100 SUSTAIN % will take approx 0.4s to get to 100
```

DELETE delete word **Command**
string -> .

DELETE removes the named user word, freeing its space. All dynamic players and sounds are stopped, and the static player's actions are reset.

DELETE should be used with care as its action cannot be reversed.

DURATION wait for a period of time

number -> .

DURATION makes the player's sounds wait for the specified number of timebase ticks before continuing. With the normal timebase period, each unit corresponds to 10 millseconds. The number must be in the range -32768 to 32767.

If the player has fallen behind time for any reason (for example, it is catching up after a FAST), some or all of the DURATION ticks will be used immediately in paying off the debt. This makes sure that temporary disruptions to music never cause players to get out of step. An accumulated debt may be cleared by FLUSH to make sure the next DURATION causes the full wait.

Negative durations are used to create a debt artificially.

```
Example:     "strike" <  % strike envelopes every 20 ticks
             REP( ON GATE 20 DURATION )REP >
```

E play ascending E

'E' plays the note E above the last note.

See A for more information.

e play descending E

'e' plays the note E below the last note.

See A for more information.

EBIG set normal/magnified state of pitch envelope
flag -> .

EBIG sets the scaling factor of the current envelope for pitch:

```
OFF EBIG     x1 (1 envelope unit = 1/16th semitone)
ON EBIG      x4 (1 envelope unit = 1/4th semitone)
```

Changes take effect on the next segment start.

The scaling factor has no effect on envelopes used for amplitude.

Normal scaling is set up by ESECT

```
Example:     % +_ one ocatave sweep
             2 EMOD          % select envelope 2 for modification
             50 48 CYCLE     % 1s period, 1/4 octave depth
             ON EBIG         % select x4 scaling
```

EDIT **put text of word in buffer** **Command**

string -> .

The text of the named user word is put in the buffer, replacing the existing contents. The lines are numbered 10 apart starting with line 10.

Note that line numbers are not stored in words, but are created from scratch by EDIT.

Example: "part1" EDIT

EGRAD **set envelope segment gradient**

differencenumber timenumber segnumber -> .

EGRAD sets the gradient of the numbered segment in the current envelope.

The gradient is specified as a fraction: the change in the value of the envelope (the first number) divided by the time taken to do it (the second number). The value change need not have anything to do with the actual change that takes place, but serves only to describe the gradient conveniently.

The value is in normal units of amplitude or pitch, with a range of -255 to +255. The time is in centisecond units with a range of -127 to +127.

Very small gradients can be used to create static segments in which the value is constant for a certain period of time. The target level should be one unit greater that the previous target, and the gradient value change should be one, whereupon the duration of the constant value will be equal to the gradient time.

The segment identified by the top number must exist in the envelope.

Examples:

```
 127 20 1 EGRAD % set segment 1 to 127 units in 20cs (0.2s)
    1 1 2 EGRAD % set segment 2 to 1 unit in 1cs (100 units/sec)

 % pitch envelope to fall slowly after note start
 2 EMOD                        % select envelope 2 for mod
 1 1 0 ESECT                   % one final segment
 32 100 1 EGRAD  -127 1 ELEV   % fall by 32 units per second
      0 2 EGRAD  -127 2 ELEV   % stay at bottom during repeat
```

ELEV **set envelope segment level**

levelnumber segnumber -> .

ELEV sets the target level of the numbered segment in the current envelope.

The target value of a segment is the value that the envelope must reach before going on to the next segment. If the envelope is already at or past the target when it enters a segment, it moves on to the next one immediately.

Note that because of this, if section 2 (the repeating section) has only one sement, the slope will act when the section is entered (assuming the envelope value is before the target) but will have no effect when repeated. This is used by ADSR-created envelopes to produce the decay and sustain from just one segment.

The value is in normal units of amplitude or pitch, with a range of -127 to +127.

The segment identified by the top number must exist in the envelope.

Examples:

```
 127 1 ELEV % set segment 1's level to 64

 % 'bend' up one semitone to note pitch at note start
 2 EMOD                        % select envelope 1 for mod
 2 1 0 ESECT                   % use two segments for bend
 -16 1 1 EGRAD   -16 1 ELEV    % down 16 units in 1cs
 16 20 2 EGRAD     0 2 ELEV    % rise to pitch in 20cs (0.2s)
```

)ELSE(separate conditional sections

IF(...)IF and IF(...)ELSE(...)IF enclose words which are to be executed conditionally.

See IF(for more information.

EMOD select envelope for modification

number -> .

EMOD selects the numbered envelope as the current envelope for modification.

The current envelope is the one that envelope words such as ESECT, ADSR, CYCLE etc. work on, and it must be set by EMOD before these words are used.

There are 13 redefinable envelopes numbered 1 to 13.

```
Example:     1 EMOD       % select envelope 1 for modification
```

ESECT set up envelope sections

onnumber repnumber offnumber -> .

ESECT sets the number of segments in each of the three sections of the current envelope and sets the scale to normal (x1).

It is used to set up the current envelope for creating a new shape with the envelope segment words; EGRAD, ELEV and EBIG.

The three numbers are the number of segments required in each section respectively.

Each segment is straight line portion of the envelope, with programmable gradient and target level. The segments are arranged in sections which determine what will happen in response to envelope gates. The three sections are:

1	On	started when the gate goes on
2	Repeat	repeated after the On section until the next gate
3	Off	started when the gate goes off

When the Off section has finished, the envelope runs into a preset 'parking' segment which holds the value constant until the next gate.

The number of segments allowed in each section and in total are as follows:

On	0 to 10
Repeat	1 to 10
Off	0 to 10
Total	1 to 10

The On and Off sections can have zero segments, giving slightly different effects in the two cases.

The zero On section is skipped over immediately so when an on gate occurs, the envelope goes straight into the repeat section.

If the Off section has zero segments, gate offs have no effect so the envelope stays in the repeat section until the next gate on.

Examples:

```
 1 1 1 ESECT  % one segment in each section

 1 2 0 ESECT  % repeat until next gate on, ignoring gate off

 0 2 0 ESECT  % go straight into repeat section
```

EVEN   leave 'even channels' group identifier
        . -> number

EVEN is used as the argument of CHAN to select all the even-numbered channels on the current voice.

```
Example:     EVEN CHAN  0 AMP      % even channels not sounding
```

F   play ascending F

'F' plays the note F above the last note.

See A for more information.

f   play descending F

'f' plays the note F below the last note.

See A for more information.

FAST   advance time
        number -> .

FAST moves the timebase forward by the specified number of ticks, making the program (all players) run through the equivalent period of time at maximum speed.

The fast ticks take effect even if the timebase is in a frozen state (after ON FREEZE).

The range allowed is 0 to 16383 ticks.

```
Example:     192 16 #* FAST  % skip 16 4/4 bars
```

FLUSH   clear waiting sounds and reset time

Any sound requests that are waiting to play on the player are cleared, and the player's record of time is reset so that the next sound will play immediately. The sound that is playing when FLUSH is executed is not affected.

FLUSH is done automatically by GO and the keyboard interpreter (when a line is entered), and will not be used explicitly in most programs.

```
Example:     "warning-tone" <   % plays immediately
             FLUSH warningsound
             ON GATE 100 DURATION OFF GATE >
```

FOR(start definite loop **↔ only**

number -> .

FOR(...)FOR encloses words that are to be executed a definite number of times.

FOR(takes a number which determines the number of times the contents of the loop are executed. If this is less than one, the contents are not executed.

FOR(and)FOR can only be used inside words.

```
Examples:   "stars" ← FOR( "*" $OUT )FOR →
                      % 5 stars   prints   *****
                      % 0 stars   prints nothing

            phrase1 8 FOR( phrase2 )FOR phrase3
                      % part of a word, with phrase2 repeated

            "scale" ← SCORE -2: 8,    % play scale over
            4 FOR( CDEFGAB )FOR ↑ →   % four octaves
```

)FOR end definite loop **↔ only**

FOR(...)FOR encloses words that are to be executed a definite number of times.

See FOR(for more information.

FREEZE start/stop timebase

flag -> .

FREEZE controls the timebase:

```
ON FREEZE    stop timebase
OFF FREEZE   allow timebase to continue
```

While the timebase is stopped, all durations last indefinitely so music is frozen, though the timebase may still be advanced by FAST. Envelopes are not affected.

The timebase is automatically unfrozen when an error occurs.

```
Example:    % function keys to hold and resume music
            *KEY4 ON FREEZE!M
            *KEY5 OFF FREEZE!M
```

FM   set depth of frequency modulation

number -> .

FM sets the depth of frequency modulation of the current channel, which must be odd, by the even channel of the same pair.

The frequency of the odd channel is proportionally modulated by the even channel's auxiliary output signal. This is a two-state sign-only version of its main signal. The FM depth can range from 0 to 255, with 192 corresponding to 100% modulation.

The timbre of the modulated signal depends on the FM depth, the waveforms, and most importantly, the frequency ratio or pitch difference. Simple intervals produce sounds with harmonic components, like those of simple vibrating objects such as strings and air columns. More discordant intervals produce sounds with non-harmonic components, characteristic of complex vibrating objects such as bells, gongs and chimes.

Note that the shapes of geometrically-created waveforms may not be 50% positive and 50% negative and will therefore give a net pitch shift when used by the modulating channel.

The FM depth is set to zero by SOUND.

```
Example:     "fmsound" <
             2 CHANS SOUND
             1 CHAN 100 FM
             2 CHAN -271 SHIFT 3000 OFFSET
             >
```

G play ascending G

'G' plays the note G above the last note..

See A for more information.

g play descending G

'g' plays the note G below the last note.

See A for more information.

GATE set envelope states

flag -> .

GATE controls the current channel's pitch amd amplitude envelopes by setting them to the starts of their On or Off sections. It takes a flag that determines the sense of the gate:

ON GATE	set gate on: move to start of On sections
OFF GATE	set gate off: move to start of Off sections

Normal amplitude envelopes turn the sound on in the On section and off in the Off section, so the on and off gates correspond to notes and rests. GATE is used by most NOTE and REST playing actions.

GO start all players together

All the players are set going at the same time, guaranteeing that the first note of each part is aligned with the others.

GO should be used after creating the players with PLAYERS and doing a PLAY(...)PLAY for each one. It cannot be used inside a player.

```
Example:    "play" ←
            0 VOICES
            2 PLAYERS
            1 PLAY( ins1 SCORE part1 )PLAY      % player 1 ready
            2 PLAY( ins2 SCORE part2 )PLAY      % player 2 ready
            GO →                                % start together
```

GVAR leave address of location for global variable ↔ only
. -> address

GVAR leaves the address of a two-byte location reserved for use as a variable. It is normally used by itself inside ← ... → to create a simple named variable.

GVAR variables are global to all players, so a value stored by one player can be read by any other.

```
Examples:   "globaltran" ← GVAR →
```

#IN wait for and get keypress
. -> number

#IN waits for a character from the keyboard and returns its code. If there is a 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 →
```

$IN input line from keyboard ↔ only
. -> string

$IN accepts a line of characters from the keyboard, terminated by carriage return. The carriage return is included as the last character of the string.

The DELETE key removes the last character typed. CTRL-U clears the line but leaves in on the screen. Other control codes are ignored.

```
Example:     % input number: . -> number ON  or  . -> OFF
             "NIN" ← $IN VAL $2 →
```

IDLE   pass control to other players

IDLE passes control to other players allowing them to continue execution.

It is used in words which wait for an external event before continuing, where it is included in the wait loop so that other players are not held up. AMPLE words that expect to be delayed by external events (including sound words, #IN and $IN) idle automatically.

Example:

```
% hold music while TAB is down
"TAB-hold" ← PLAY(                              % definition
REP( REP( IDLE -97 OKEY )UNTIL( )REP           % idle until key down
     ON FREEZE
     REP( IDLE -97 OKEY NOT )UNTIL( )REP % idle until key up
     OFF FREEZE
)REP )PLAY →

4 TAB-hold                           % use it on spare player, here 4
```

IF(start conditional   ↔ only

flag -> .

IF(...)IF and IF(...)ELSE(...)IF enclose words which are to be executed conditionally.

IF(takes a flag and tests it. In the case of IF(...)IF, the enclosed sequence is executed if the flag was ON. Where an)ELSE(is included, the words up to the)ELSE(are executed if the flag was ON, and the words between)ELSE(and)IF are executed if the flag was OFF.

IF(,)ELSE(and)IF can only be used inside words.

```
Example:     "test" ← IF( "ON" )ELSE( "OFF" )IF $OUT →
                               % ON test    prints   ON
                               % OFF test    prints   OFF
```

)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**
. -> 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.

The FOR(...)FOR and INDEX must be in the same word, that is, the INDEX cannot be inside a word inside the loop.

INDEX can only be used inside words.

```
Examples:    "countdown" <-  % print numbers from 20 down to 1
             20 FOR( INDEX NOUT SP )FOR NL ->

             4 VOICES                  % part of word,
             4 FOR( INDEX VOICE        % to set up 4 voices for
                    instrument )FOR    % homogenous chords
```

INVERT set inversion state
flag -> .

INVERT sets the sense of the channel's waveform:

```
OFF INVERT   normal
ON INVERT    inverted
```

The inverted waveform sounds the same as the normal one when playing alone, but when combined with another it can sound subtly different. The effect is particularly noticeable when the same waveform is used at slightly different frequencies. Special stereo effects can be created by positioning the channels apart.

SOUND sets the normal state.

```
Example:     "tremsound" <-           % tremelo/chorus effect
             SOUND
             2 CHAN 120 AMP           % amp controls depth
             ON INVERT  800 OFFSET    % offset controls rate
             ->
```

K(start key signature

K(...)K sets the key signature for the player, setting pitch modifications to be applied to the following notes.

The effect of + and - words used on a note inside K(...)K is applied to every occurrence of that note in the following music, except where +, - or = is applied directly to the occurrence.

K(clears existing modifications. Notes inside key signatures do not play (do not execute the NOTE playing action) or alter the effective last note.

Parallel note groups and key signatures are not allowed inside key signatures.

Examples:	K( )K	C major
	K(+F +C +G)K	A major
	K(-B)K	F major

)K   end key signature

K(...)K sets the key signature for the player, describing pitch modifications to be applied to the following notes.

See K(for more information.

LEN   get length of string **<-> only**
string -> string number

The length of the string is found, preserving the string.

```
Example:     LEN 0 #=     % test if string is null (inside word)
```

LIST   display text **Command**

The contents of the text buffer are listed on the screen.

Each line appears in input format (with line number and dot) so that it can be edited and re-entered using the cursor keys and COPY.

LOAD   load file **Command**
string -> .

The named file is loaded as user words and text, replacing existing user words and and text. All dynamic players and sounds are stopped, and the static player's actions are reset.

The file must have been created by SAVE.

```
Example:     "example" LOAD
```

MAX leave greatest of two numbers

number1 number2 -> largestnumber

The greatest of the two numbers is left and the other is discarded.

Example: -5 -2 MAX goes to -2

MEM show memory usage in bytes **Command**

MEM shows the number of bytes used by words, text and players, and the total number of bytes free.

Note that the free memory may be fragmented so the largest single piece may be smaller than the total figure given.

```
Example:     %MEM
             1663 words, 37 text, 0 players, 10337 free
```

MIN leave least of two numbers

number1 number2 -> smallestnumber

The least of the two numbers is left and the other is discarded.

Example: 4 5 MIN goes to 4

MODE enter specified display mode **Command**

number -> .

The specified display mode is entered. Free memory is first compacted, stopping dynamic players and sounds and resetting the static player's actions.

Example: 6 MODE % enter to mode 6

NEW discard user words **Command**

All user words are discarded and their space made available for re-use. The text buffer is not affected. All dynamic players and sounds are stopped, and the static player's actions are reset.

NEW should be used with care as its effect cannot be reversed.

NL print new line

Carriage return/line feed is sent to the screen, moving the cursor to the start of the next line.

NL calls OSNEWL.

NOT invert sense of flag

flag1 -> flag2

The flag is replaced by its opposite sense i.e. ON is replaced by OFF, OFF is replaced by ON.

Note that NOT is not a bitwise operator.

```
Example:     #< NOT       % ON if number was greater than or
                          % equal to previous number
```

NOTE(start note playing action <-> only

NOTE(...)NOTE selects the enclosed sequence of words as the playing action for the player's notes.

The playing actions are the actions that take place when a music words such as notes, rests or ties are executed. They determine how these musical events are interpreted.

The music words are entirely self-contained as a group and their only output is through the playing actions, which contain the sound words that play the music. Each note, rest and tie calls the appropriate playing action with data giving a description of the event. This includes the length of the previous event, since each event is resposible for the preceding duration.

The NOTE(...)NOTE sequence is called by notes, and is supplied with three numbers:

```
. -> pitch musicvoice prevnotelength     % as seen by the action
```

pitch	the pitch determined by the note name, accidentals and keysignature
musicvoice	the voice number set by ';'
prevnotelength	the length of the previous event

The default playing action for notes is:

```
DURATION     % wait for duration of prevoius note
VOICE        % select sound voice indicated by music voice
PITCH        % set note pitch
ON GATE      % turn envelope gates on
```

This would be set by:

```
NOTE( DURATION VOICE PITCH ON GATE )NOTE
```

The NOTE playing action is reset to this by SCORE.

Examples:

```
        "transposition" ← % minus one tone
        NOTE( DURATION VOICE 32 #- PITCH ON GATE )NOTE →

        % use one channel per music voice
        "dubvoice" ←        % voice selecting word.  number -> .
        1 #+ 2 #/ #12       % leave channnel-1 under voice-1
        VOICE 1 #+ CHAN → % select voice and channel

        "double" ←          % set up instrument action
        NOTE( DURATION dubvoice PITCH ON GATE )NOTE
        REST( DURATION dubvoice OFF GATE )REST
        →
```

)NOTE end note playing action ↔ only

NOTE(...)NOTE selects the enclosed sequence of words as the playing action for the player's notes.

See NOTE(for more information.

NOUT print number in decimal
number -> .

The number is printed on the screen in signed decimal representation without formatting spaces.

```
Examples:   56 NOUT   prints   56

            "ppitch" ← "Pitch: " $OUT NOUT NL →
            % 32 ppitch   prints   Pitch: 32
```

&NOUT print number in hex
number -> .

The number is printed on the screen in unsigned hex representation without formatting spaces

```
Examples:   255 &NOUT   prints   FF

            "pregs" ←
            NL "PA: &" $OUT &NOUT
              " YX: &" $OUT &NOUT →

            % &FF00 &33CA pregs   prints   PA: &33CA YX: &FF00
```

ODD leave 'odd channels' group identifier
. -> number

ODD is a constant for use as a group identifier with CHAN. ODD

CHAN selects all the voice's odd channels.

Example: ODD CHAN ON RM

OFF   leave false flag
```
       . -> number
```

OFF leaves 0, representing the 'false' flag value.

It is used with commands and other words that take a flag argument, and in logical expressions.

OFF is also used as the 'no channels' group identifier for CHAN. OFF CHAN deselects all channels of the current voice so that channel-specific commands are ignored.

```
Examples:     OFF PRINT            % turn printer off
              flagvar #? OFF #= % equivalent to   flagvar #? NOT
```

OFFSET set frequency offset
```
       number -> .
```

OFFSET adjusts the frequency of the channel by a constant amount which does not depend on the pitch.

Small frequency differences between the channels of a voice are used to enrichen the sound, and with modulation to create cyclically-varying tones. They can also be used to detune voices against each other for more realistic ensemble effects. Extreme offsets can be used with changing pitches for special frequency modulation effects.

The number supplied to SHIFT is the frequency offset in 0.0056Hz units, with a range of -32768 to 32767.

SOUND sets the offsets of all channels to zero.

```
Example:     1 CHAN 0 OFFSET
             2 CHAN 200 OFFSET  % detune pair
```

ON   leave true flag
```
       . -> number
```

ON leaves -1 representing the 'true' flag value.

It is used with commands and other words that take a flag argument, and in logical expressions.

Examples: ON FREEZE

OR **OR bits of number with bits of previous number**
number1 number2 -> number3

Each bit of the result is the logical OR of the corresponding bits of the two numbers. OR acts as a flag operator if both the numbers are flags (ON or OFF).

```
Examples:    1 OR                      % set bit 0 of number
             #11 -1 #= #12 1 #= OR     % number -> flag
                                       % test number was 1 or -1
```

#OUT **send ASCII code to screen**
number -> .

The number is sent to the screen via OSWRCH.

```
Example:     12 #OUT              % clear text screen
```

$OUT **print string** **↔ only**
string -> .

The string is printed on the screen using OSASCII.

```
Example:     "hello" $OUT   inside a word, prints   hello
```

$PAD **pad string with spaces** **↔ only**
string1 number -> string2

Spaces are added to the start (left end) of the string to make it up to the length indicated. If it is not less than this length, it is left unchanged.

```
Examples:    "hello" 8 $PAD   goes to   "   hello"

             % print number in field.  number fieldnumber -> .
             "nfout" ← #12 $STR $PAD $OUT →
             20 4 nfout 10 4 nfout   prints   ~sp~sp20~sp~sp10
```

PAGE **set page mode state** **Command**
flag -> .

PAGE controls the display's page mode:

ON PAGE	turn page mode on
OFF PAGE	turn page mode off

When page mode is on, scrolling text waits for the SHIFT key to be down before printing the next page.

```
Example:     ON PAGE WRITE        % display text page by page
```

PAIR leave specified 'pair of channels' group identifier
number1 -> number2

PAIR converts a channel number into the group identifier of the corresponding channel pair for use by CHAN. PAIR CHAN takes a channel number and selects that channel and the other channel of the same pair on the current voice.

PAIR is often used to control the two channels of a modulating pair in parallel.

```
Example:     1 PAIR CHAN 0 SHIFT         % play pairs
             3 PAIR CHAN 112 SHIFT       % in fiths
```

PENV select pitch envelope
number -> .

PENV selects the numbered envelope for use by the current channel as its pitch envelope.

The pitch envelope of a channel controls the pitch (along with PITCH, TUNE and SHIFT), varying it within the note.

The number is an envelope number in the range 1 to 10, or an envelope identifier left by SIMPLEP (or SIMPLEA).

Newly selected envelopes are at zero until they receive a gate.

```
Examples:    2 PENV            % select envelope 3
             SIMPLEP AENV      % restore default envelope
             vibraP #? AENV    % select envelope from variable
```

PGATE set pitch envelope state
number -> .

PGATE controls the current channel's pitch envelope by setting it to the start of the On or Off section. It takes a flag that determines the sense of the gate:

```
ON PGATE      set gate on: move to start of On section
OFF PGATE     set gate off: move to start of Off section
```

PGATE is used to control the pitch envelope independently of the amplitude envelope.

PITCH set pitch
number -> .

PITCH sets the base pitch of the current channel. The sum of the base pitch and the SHIFT, TUNE and pitch envelope values gives the channel's playing pitch.

PITCH is used by notes (via the NOTE action) to set the playing pitch of notes.

Pitch is described in 16th semitone units, so a semitone is 16 and an octave is 192. Middle C has a value of 0, and the range of pitch is a -1024 to 1023.

SOUND sets the value of PITCH to zero.

Example:

```
"hollycomp" ← % random pitch sequence
2 CHANS SOUND           % plain sound
REP( RAND?              % get random number
     193 #/ #12 #2      % make it the range 0 to 192 (one octave)
     384 #+             % move up into third octave above mid C
     PITCH              % play random pitch
     ON GATE            % strike
     10 DURATION        % wait for 10th second
)REP →
```

PLAY(start concurrent sequence **↔ only**
number -> .

PLAY(...)PLAY encloses a sequence of words to be executed concurrently with (alongside) other sequences.

PLAY(takes the number of the existing dynamic player which is to execute the sequence. If the player is doing nothing when it receives the PLAY request, it starts execution on the next GO (or the second call of IDLE). If it is occupied, execution starts when the current and pending PLAY sequences have been finished.

PLAY(and)PLAY can only be used inside words.

Examples:

```
"play" ←                    % piece for two players
0 VOICES 2 PLAYERS
   1 PLAY( ins1 READY SCORE part1 )PLAY
   2 PLAY( ins2 READY SCORE part2 )PLAY
GO →

"canvas" ←          % play background notes
1 PLAYERS           % interactive sound editing
1 PLAY(
  SOUND SCORE 16,
  REP( -2: 4 FOR( CEGB )FOR )REP
     )PLAY →
```

)PLAY   end concurrent sequence **↔ only**

PLAY(...)PLAY encloses a sequence of words to be executed concurrently with other sequences.

See PLAY(for more information.

PLAYERS   create players
number -> .

PLAYERS creates the specified number of dynamic players. It stops all dynamic players and sounds, and resets the static player's playing actions.

When a newly-created player is first used, it is automatically given one voice with two channels, and this voice is selected.

Up to and including eight dynamic players can exist. Note that for eight dynamic players to be created with one voice each, all voices on the static player must be freed by 0 VOICES.

Example: 3 PLAYERS

PNUM   leave player number
. -> number

PNUM leaves the number of the player in which it is executed. It is primarily used to identify players to each other for sharing of resources.

PNUM returns zero in the static player.

POS   set stereo position
number -> .

POS sets stereo position of the channel.

The range is -3 (maximum right) to 3 (maximum left) with 0 being the central position. SOUND sets the position to centre.

Example:
```
ON CHAN -3 POS   % put all channels at maximum left

"mobile" ←
SOUND
REP( RAND? 4 #/ #12 #2 POS          % random position
     ON GATE 10 DURATION
     OFF GATE 10 DURATION
)REP →
```

PRINT   set printer state **Command**

```
        flag -> .
```

PRINT controls the printer. When the printer is turned on, all text sent to the screen is also sent to the printer.

```
ON PRINT    turn printer on
OFF PRINT   turn printer off
```

PVAR leave address of location for player variable ↔ only

```
        . -> address
```

PVAR leaves address of a reserved two-byte location specific to the player. It is normally used by itself inside ← ... → to make a simple named variable which has a separate location for each of up to nine players.

PVAR variables are used in words that need to run simultaneously but independently in a number of players.

```
Example:     "localtran" ← PVAR →
```

QKEY test key status/get key

```
        negative number -> flag
        zero   -> number
```

QKEY tests whether a key is down, or gets a character from the input buffer.

Given a negative number, QKEY tests the key identified by the number, giving the answer ON if the key is down and OFF if it is not. See the BBC Micro User Guide under 'INKEY' for a list of the negative key numbers.

Given zero as the number, QKEY returns a character from the the keyboard or if there is no character, a negative number.

QKEY should not be used with numbers greater than 0.

```
Example:     "TAB-state" ← -97 QKEY →
```

RAND? get random number

```
        . -> number
```

RAND? produces a random number in the range -32768 to 32767.

```
Example:     RAND? #0<      % give random flag - ON or OFF
```

RAND! set starting point for random numbers

```
        number -> .
```

RAND! sets the random number seed. For each value set by RAND!, RAND? generates the same sequence of numbers.

The seed is normally the last number generated by RAND?, so setting it to a number taken from a random sequence will re-start the sequence from that point.

```
Example:     restart #? RAND!  % re-start particular sequence
```

RELEASE **set release time of ADSR envelope**
number -> .

RELEASE sets the release time of the current envelope, which will normally have been set up by ADSR.

The release is the fall of the amplitude to zero when the envelope gate turns off. This happens when a note finishes assuming that another note does not start, that is, that it is followed by a rest.

RELEASE sets the time, in centisecond units, that it takes to go from maximum (127 units) to zero. The range is 1 to 32767.

Note that if a lowered sustain level is used (a level less than 127) the release time will be correspondingly shorter.

```
Example:     1 EMOD       % select envelope 1 for modification
             ADSR         % set up basic ADSR envelope
             200 RELEASE  % fade over 2s during rest
```

REN **renumber text** **Command**

REN renumbers the lines in the text buffer, so that the first line is line 10 and successive lines are 10 apart.

REP(**start 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.

The simple REP(...)REP loop executes the contents repeatedly until ESCAPE or STOP.)UNTIL(is included to exit the loop conditionally. It takes a flag and if it is ON, jumps to the first word after)REP. Up to 30)UNTIL(s are allowed.

REP(,)UNTIL(and)REP can only be used inside words.

```
Examples:    "forever" <- REP( O:CGd-E Fc/b )REP ->

             "TABwait" <- % wait for TAB press
```

```
REP( #IN 9 #= )UNTIL( )REP →
```

)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.

REPORT report error location **Command**

REPORT shows where in the program the last error occured. It prints the numbered line with a ! indicating the AMPLE word that generated the error.

The line number that appears is the number that the line will have when the word is converted to text by EDIT. If the word's text is already in the buffer, the line may have a different number. To avoid confusion, use EDIT on the word and correct the new text.

REPORT's record of the last error in a user word is not affected by errors in the input line and AMPLE words used as commands.

REPORT does nothing if, since the error, a command such as DELETE or LOAD was used that could have removed the user word.

```
Example:      %play                      % run user word
             No number in instr1        % error message
              %RPEORT                    % typing error
             Mistake                    % error message
              %REPORT                    % command
                30.2 CHAN AMP           % line printed by REPORT
                           !            % indicates AMP caused error
              %                          % prompt
```

REST(start rest playing action **↔ only**

REST(...)REST selects the enclosed sequence of words as the playing action for the player's rests.

The playing actions are the actions that take place when a music words such as notes, rests or ties are executed. They determine how these musical events are interpreted.

The music words are entirely self-contained as a group and their only output is through the playing actions, which contain the sound words that play the music. Each note, rest and tie calls the appropriate playing action with data giving a description of the event. This includes the length of the previous event, since each event is resposible for the preceding duration.

The REST(...)REST sequence is called by ↑, and is supplied with two numbers:

```
musicvoice prevnotelength -> .           % as seen by the action

musicvoice               the voice number set by ';'
prevnotelength           the length of the previous event
```

The default playing action for rests is:

```
DURATION     % wait for duration of prevoius note/rest/tie
VOICE        % select sound voice indicated by music voice
OFF GATE     % turn envelope gates off
```

This would be set by:

```
REST( DURATION VOICE OFF GATE )REST
```

The REST playing action is reset to this by SCORE.

)REST   end rest playing action   ↔ only

REST(...)REST selects the enclosed sequence of words as the playing action for the player's rests.

See REST(for more information.

$REV   reverse the order of characters

string -> reversed-string

The order of the characters in the string is reversed.

Examples: "hello" $REV   leaves   "olleh"

RM   set ring modulation state

flag -> .

RM sets the state of ring modulation of the current channel, which must be odd, by the even channel of the same pair. The flag determines the state:

```
ON RM    turn ring modulation on
OFF RM     turn ring modulation off
```

The odd channel's signal is multiplied by the even channel's auxiliary output signal. This is a two-state sign-only version of its main signal.

The timbre of the modulated signal depends on the waveforms, and the frequency ratio or pitch difference. Simple intervals such as unison and octave produce sounds with harmonic components, like those of simple vibrating objects such as strings and air columns. The timbre is bright, and dependant on the relative phase so a small frequency offset produces continuouly changing timbres.

More discordant intervals produce sounds with non-harmonic components. These can be very dense and abrasive. With very complex waveforms, ring modulation produces pseudo-random noise whose tone and density are dependant on the oscillator frequencies.

SOUND performs OFF RM.

```
Example:     "rmsound" ← 2 CHANS SOUND
             1 CHAN ON RM
             2 CHAN 20 OFFSET
             →
```

SAVE **save file** **Command**

string -> .

The user words and text are saved as the named file. Free memory is compacted, all dynamic players and sounds are stopped, and the static player's actions are reset.

If the user words form a complete program, it is usual to put just the name of the main word (and a comment) in the text buffer so that RUN will run the reloaded program.

If only user words need be saved, CLEAR should be executed before saving to remove text that would otherwise unnecessarily add to the file size.

```
Example:     %CLEAR                        % discard old text
             %10.% Pulstar                 % title
             %20.% 15 November 1983        % date
             %30.play                      % name of main word
             %"pulstar" SAVE               % save ready to run
```

(The % at the start of each line is the prompt, not part of the input line.)

SCAN **repeat input line with variable number** **Command**

SCAN accepts a line of words and then interprets single-key commands which execute the line. A single number is supplied to the line and displayed on the screen. The command keys allow this value to be altered on each execution of the line.

SCAN is mainly used for experimenting with sound words, sweeping the value of a parameter to hear its effect. The words on the line must use the number provided, that is, have the effect:

```
        number -> .
```

The eight command keys are grouped at the right of the keyboard. Six keys increment or decrement the number by certain amounts, and one key sets it to zero. They all execute the line with the new value. Pressing the space bar repeats the line with the same value. RETURN leaves SCAN and returns to the % prompt.

The scanned line must consume the number supplied.

```
Examples:   SCAN PITCH                % vary pitch in steps

            SCAN VOICE instrument     % set up successive
                                      % voices by pressing →
```

SCORE   prepare for new music

SCORE resets the player's music word variables to initial values. It is used at the start of each scored part of a piece to prepare for music words.

The initial state of music is:

```
        48,  1;  0:  0 BAR
        C  |
```

SCORE also resets the NOTE, REST and TIE actions to their defaults.

```
Example:    SCORE  K( +F )K  192 BAR     % signature
```

SHARE   use specified player's voices
number -> .

SHARE sets the number of the player whose voices are to be used by channel-specific sound commands issued by the current player.

VOICES, CHANS and SOUND affect the shared group of voices, but each player selects voices and channels for control (using VOICE and CHAN) independently of other players using the same group.

When a player is created, it initially uses its own group of voices.

Examples:

```
        % play-time sound editing
        1 SHARE 1 VOICE 120 AMP      % alter voice 1 of player 1
```

```
PNUM SHARE                        % revert to own voices

% dynamic voice assignment

"dyvo" ← 0 VOICES 2 PLAYERS
 1 PLAY( 6 VOICES                 % owns 6 voices
 6 FOR( INDEX VOICE SOUND )FOR    % sets them up
 SCORE 1;F(ACFA) C                % uses no. 1, shares 2-5
        ↑(↑↑↑↑↑)
   )PLAY
 2 PLAY( 1 SHARE                  % borrows 6 voices
   SCORE 6;C 2;E(GCEG)            % uses no. 6, shares 2-5
   )PLAY
 GO →
```

SHIFT set pitch shift
number -> .

SHIFT determines what pitch the channel will play at in relation to the pitch of the note set by PITCH. It sets a pitch offset which is added to the base pitch and the TUNE and pitch envelope values to give the final channel pitch.

SHIFT is used in complex sounds to play different channels of a voice at harmonically related pitches, and for special musical effects like parallel harmonies and chords.

The number is the offset in 16th semitone units from middle C, with a range of -1024 to 1023. SOUND sets the value to zero.

SHOW show user words **Command**

SHOW displays the names and the total number of all user words currently defined.

Example:

```
sync     act      play     riff2
riff     start    next     wait
all
9 words
```

SIGN test number is negative
number -> flag

A flag is left which is ON if the top number was negative, and OFF if it was zero or positive.

SIMPLEA leave identifier of simple amplitude envelope
. -> number

SIMPLEA is the default amplitude envelope selected by SOUND.

The envelope has an immediate attack and a fast release. There is a short deacy to a slightly lowered sustain level to highlight the start of each note and separate successive notes of the same pitch.

SIMPLEA is used with AENV to restore the default amplitude envelope on the current channel.

```
Example:     SIMPLEA AENV         % select SIMPLEA
```

SIMPLEACT   select simple playing actions

SIMPLEACT selects the default playing actions for the player.

It is used to restore normal actions after changing one or more with NOTE(...)NOTE, REST(...)REST or TIE(...)TIE.

```
Example:     SIMPLEACT   % restore default actions
```

SIMPLEP   leave identifier of simple pitch envelope
. --> number

SIMPLEP is the default pitch envelope selected by sound.

The shape of the envelope is flat in all sections, so it has no effect on the pitch of the note.

SIMPLEP is used with PENV to restore the default pitch envelope on the current channel.

```
Example:     SIMPLEP PENV         % select SIMPLEP
```

SIMPLEW   leave identifier of simple waveform
--> number

SIMPLEW is the default waveform selected by sound.

SIMPLEW is used with WAVE to restore the default waveform on the current channel.

```
Example:     SIMPLEW WAVE         % select SIMPLEW
```

SOUND prepare for new sound

SOUND resets all parameters on all channels on the current voice to initial states. The amplitude of channel 1 is set to maximum and the amplitude of all other channels is set to zero.

It is used at the beginning of each instrument definition, and by itself sets up a simple instrument. It should be used to initialise each voice after assignment, either directly or as part of an instrument definition.

SOUND leaves channel 1 selected.

The initial state of all channels set up by SOUND is:

```
 0 PITCH  OFF GATE
 0 SHIFT  0 OFFSET  0 POS  0 INVERT
 0 WAVE SIMPLEP PENV  SIMPLEA AENV
 OFF RM  OFF SYNC  0 FM

 Channel 1: 128 AMP
 Other channels: 0 AMP
```

SOUND cannot be used on a voice with no channels.

```
Example:     "simple" <
             SOUND           % set all voice's sound values
             1 WAVE
             2 AENV >
```

SP   print a space

```
Example:     5 NOUT SP -4 NOUT   prints 5 -4
```

STOP   stop players and sounds

STOP's action is as follows:

 1 stop all sounds
 2 stop and discard dynamic players
 3 unfreezes the timebase (NO FREEZE)
 4 set the static player's note context to normal
 5 set the static player to use its own voices (PNUM SHARE)

STOP is used as a command to stop a piece which is running on dynamic players, and in words to end execution immediately and return to the keyboard.

```
Examples:     %RUN                    % command to start playing
              %STOP                   % end piece prematurely

            #IN 13 #= IF( STOP )IF % in word, to end on RETURN
```

$STR   convert number to decimal string representation   ↔ only
       number -> 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.

Example: -425 $STR   goes to   "-425"

&$STR   convert number to hex string representation   ↔ only
number -> string

&$STR converts a numbre to the string of characters representing the number in hexadecimal.

Example: 254 &$STR   goes to   "FE"

$STRIP remove leading spaces   ↔ only
string1 -> string2

Any spaces on the left end of the string are removed.

$STRIP is commonly used on input strings before decoding numbers with VAL.

Example: "   hello" $STRIP   goes to   "hello"

SYNC   set synchronisation state
flag -> .

SYNC sets the state of synchronisation of the current channel, which must be odd, by the even channel of the same pair. The flag determines the state:

ON SYNC   turn synchronisation on
OFF SYNC   turn synchronicsation off

The phase of the odd channel's signal is reset to zero when the even channel's auxiliary output signal is negative or its phase passes zero. The auxiliary output signal is a two-state sign-only version of the main signal.

Simple synchronisation uses an all-positive waveform (typically all zeros) on the synchronising channel so that the synchronised channel's phase is reset only when the synchronising channel's phase passes zero.

In the case of simple synchronisation, the timbre is based on the original waveform and is always harmonic. The synchronised tone plays at the pitch of the synchronising channel, and its own pitch controls the timbre. With the synchronised pitch higher than the synchronising pitch, the sound is strongly coloured and sometimes vowel-like.

SOUND performs OFF SYNC.

SUSTAIN sets sustain level of ADSR envelope

SUSTAIN sets the sustain level of the current envelope, which will normally have been set up by ADSR.

The sustain level is the amplitude of the note once the initial attack and decay have finished, that is, the amplitude it settles at when the note is held indefinitely.

The sustain level is in amplitude units, in the range 0 to 127.

```
Example:     1 EMOD       % select envelope 1 for modification
             ADSR         % set up basic ADSR envelope
             100 SUSTAIN % set quit sustain level
```

TEMPO set timebase period

number -> .

TEMPO sets the period of the master timebase in 10 microsecond units.

The timebase controls all DURATIONs and therefore the tempo of music. It does not control envelopes or frequencies.

The range is 26 (3646 per second) to 65535 (1.53 per second).

The initial value is 1000 (100 per second).

```
Example:     500 TEMPO          % double normal tempo
```

TIE(start tie playing action <-> only

TIE(...)TIE selects the enclosed sequence of words as the playing action for the player's notes.

The playing actions are the actions that take place when a music words such as notes, rests or ties are executed. They determine how these musical events are interpreted.

The music words are entirely self-contained as a group and their only output is through the playing actions, which contain the sound words that play the music. Each note, rest and tie calls the appropriate playing action with data giving a description of the event. This includes the length of the previous event, since each event is resposible for the preceding duration.

The TIE(...)TIE sequence is called by /, @ and ')', and is supplied with one number:

```
prevnotelength -> .     % as seen by the action
```

prevnotelength   the length of the previous event

The default playing action for ties is:

```
DURATION     % wait for duration of previous note
```

This would be set by:

```
TIE( DURATION )TIE
```

The TIE playing action is reset to this by SCORE.

)TIE   end tie playing action   **↔ only**

TIE(...)TIE selects the enclosed sequence of words as the playing action for the player's ties.

See TIE(for more information.

TIME   leave value of time
. -> number

TIME returns the period of time, in timebase units, to go before the next sound request on the player will play. As time passes TIME decreases and the next sound request plays when it goes below zero. DURATION sound requests add to the value of TIME, and FLUSH sets it to -1.

```
Example:     % wait for last DURATION to finish to synchronise
             % an asynchronous action such as screen display
             "soundsync" ←
             REP( TIME 0 #< )UNTIL( )REP → % wait for -ve TIME
```

TUNE   set tuning of synthesiser
number -> .

TUNE sets a pitch offset which applies to the whole synthesiser, allowing it to be tuned up or down to match other instruments.

The number is the offset in 16th semitone units in the range -1024 to 1023.

Its initial value is zero.

)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.

VAL **convert string to unsigned decimal number** **↔ only**

```
string -> remaining-string number ON     if found
       -> remaining-string OFF           if not found
```

The string is decoded as a decimal number, leaving the remainder. The number is left with ON on top, or if no number was found, OFF is left.

Minus signs are not recognised. Leading spaces are not ignored (they can be removed beforehand with $STRIP).

```
Example:     "10 20" VAL    goes to    " 20" 10 ON
```

&VAL **convert to unsigned hex number** **↔ only**

```
string -> remaining-string number ON     if found
 -> remaining-string OFF                 if not found
```

The string is decoded as a hex number, leaving the remainder. The number is left with ON on top, or if no number was found, OFF is left.

Leading spaces are not ignored (they can be removed beforehand with $STRIP).

```
Example:     "0F 3C" &VAL    goes to    " 3C" 15 ON
```

VERSION **return version number**

```
. -> number
```

VERSION returns a number indicating the version of AMPLE in use.

```
Example:     VERSION NOUT    prints    10    on version 1.0
```

VOICE **select specifed voice of player**

```
number VOICE -> .
```

The specified voice of the player is made the current voice, that is, the target of future voice-specific sound words. Only those voices that the player owns (have been assigned to it) can be selected.

Player's voices are numbered from 1 upwards. Voice 0 is a dummy voice which has no channels and ignores all channel-specific sound requests.

When a player is created, it has voice 1 (its only voice) selected.

```
Example:      1 VOICE instr1         % set up voices
              2 VOICE instr2         % for two part piece
```

VOICES assign voices to this player
number -> .

VOICES assigns the specified number of voices to the player in place of any it previously had. Voice 0 (the dummy voice) is made the current voice for control.

Each player, including the static player, is given one voice (with two channels) when it is created. 0 VOICES frees all the player's voices so they can be reassigned.

Each of the newly-assigned voices has two channels assigned to it.

There is a total of eight voices available in the system.

```
Examples:     0 VOICES              % free all voices on this player

              3 VOICES              % set up player for
              1 VOICE instr         % three-note chords
              2 VOICE instr
              3 VOICE instr
```

WAVE select waveform
number -> .

WAVE selects a waveform for use by the current channel.

The number is a waveform number in the range 0 to 13.

WAVE selects the waveform that will sound, and works totally independently of WMOD.

```
Example:      3 WAVE       % use waveform 3
```

WG! write point in geometric waveform
valnumber pointnumber -> .

WG! sets the value of a single point in the geometric waveform buffer.

It is used to create waveforms geometrically, that is, by specifying each point individually. This is usually done by a program that calculates the waveform points from a set of rules or parameters.

The value has a range of -127 to 127, and there are 128 points numbered 1 to 128.

The change caused by WG! does not take effect on th sound until the the current waveform has been updated by WGC.

```
Example:      % create ultra-rich waveform using RAND?
              "randwave" ←           % waveform definition
              0 RAND!                % ensure same waveform each time
              128 FOR( RAND? INDEX WG! )FOR % write all points
              WGC →                  % update current waveform
              1 WMOD randwave        % use it on waveform 1
```

WG?   read point in geometric waveform

pointnumber -> valnumber

WG? returns the value of a single point in the geometric waveform buffer.

It is used to process (with WG!) and read geometric waveforms which have been created directly or converted from harmonic form.

There are 128 points numbered 1 to 128, and the value has a range of -127 to 127.

```
Example:      % apply 'clipping' to geometric waveform

              "clipwave" ←
              128 FOR( INDEX WG?          % read point
                       2 #*               % amplify
                       127 MIN            % limit at +127
                       -127 MAX           % limit at -127
                       INDEX WG!          % write point
                  )FOR →
```

WGC   copy geometric waveform to current waveform

WGC copies the contents of the geometric waveform buffer to the current waveform (selected by WMOD).

It is used when a geometric waveform, constructed directly by WG! or from the harmonic waveform buffer by WHG, is complete.

WH!   write harmonic in harmonic waveform

ampnumber harmnumber -> .

WH! sets the amplitude of the numbered harmonic in the harmonic waveform buffer.

WH! is used to create waveforms harmonically, that is, by specifying the relative amplitude of each of the first 16 harmonics individually.

The amplitude has a range of 0 to 127. For the most accurate results, the strongest harmonic in a waveform should be at maximum amplitude (127 units).

The harmonic number has a range of 1 to 16.

The new harmonic waveform will not be heard waveform until converted to geometric form by WHG and copied to the current waveform by WGC.

Examples:

```
"skinnywave" ← % create waveform from literal data
                1 WMOD       % use waveform 1
                WZERO        % zero all harmonics
                127 1 WH!    % strong fundamental
                 60 3 WH!    % medium third
                 40 7 WH!    % medium seventh
                WHG WGC →    % finish

% using SCAN on harmonics
"harm" ← WH! WHG WGC →  % temporary word. number -> .
3 WAVE ON GATE          % turn on sound using waveform 3
3 WMOD                  % select waveform 3 for modification
SCAN 2 harm             % scan ampitude of second harmonic

"rampwave" ← % create low-pass filtered ramp wave by computation
           2 WMOD                         % use waveform 2
           16 FOR( 127 INDEX #/ #2        % amp = max/harmonic no.
                   INDEX WH! )FOR         % set amplitude
           WHG WGC →                      % finish
```

WHG   convert harmonic waveform to geometric waveform

WHG converts the contents of the harmonic waveform buffer (created with WH!), into geometric form in the geomtric waveform buffer.

The geometric waveform buffer will normally be immediately copied to the current waveform by WGC, but it can be processed using WG! and WG? before doing so.

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 spooled to a file.

A dummy definition of each word is written out at the start so that whatever the order of the definitions, the text can be sent to a file with *SPOOL and the program re-created by executing it with *EXEC.

```
Examples:     ON PRINT WRITE OFF PRINT % print program

              *SPOOL text              % spool text for editing
              WRITE *SPOOL             % with word processor
```

WMOD   set current waveform
        number -> .

WMOD sets the number of the waveform to be modified.

WGC copies the geometric waveform constructed with or WG! to the waveform selected by WMOD.

WMOD it totally independent of WAVE.

```
Example:      1 WMOD    % select waveform 1 for modification
```

WZERO   clear harmonic and geometric waveforms

WZERO sets all the amplitudes of the harmonic waveform to zero, and all the points of the gemetric wavform to zero.

It is used to prepare for creating a harmonic waveform so that harmonics that are not set with WH! are at zero. It is not necessary if all 16 harmonic amplitudes are set with WH!.

XOR   ex-OR bits of number with bits of previous number
        number1 number2 -> number3

Each bit of the result is the logical exclusive-OR of the corresponding bits of the two numbers. XOR acts as a flag operator if both the numbers are flags (ON or OFF). As a flag operator it is equivalent to #= NOT.

```
Examples:     &FFFF XOR            % complement bits in number
              fvar1 #? fvar2 #? XOR % ON if fvar1 = fvar2
```

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

Notes

ACORN COMPUTER

Acorn Computers Limited Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.