The

Cakewalk Application Language Programming Guide

for SONAR

Version: 2.2

Original author:	**D. Glen Gardenas**
Editing and adaptations:	**Ton Valkenburgh**
Review:	**Frans H.M. Bergen**

Febrary, 24, 2010

http://www.MIDI-Kit.nl

Contents

A word from the author and editor

Over the past few years, producing music on the keyboard has been synonymous with the use of sequencers, particularly PC based sequencer software. With the sophistication of off-the-shelf sequencer packages for the PC, a person needs only to connect a MIDI or serial keyboard to a computer to start producing professional demos. In fact, the software can even replace the instrument if you are willing to take the time and care to enter notes by hand and edit in the "live" feel. Do not think this cannot be done without sounding obviously unperformed. With a bit of time and a creative touch, it is amazing what can be done with nothing else then a good synthesizer, and a computer running high-powered sequencer/editor software.

The key to unleashing your creativity using any sequencer is to avail yourself to all of the tools it provides. This is true regardless of how much or how little equipment, time or experience you have. If you want the best results, you must open all of the doors. In this case, open all of the dialog boxes and menu lists. Find out what your sequencer can do and use those tools to their limit. If the music is going to sound right, the editing must be right. However, as flexible as modern PC based sequencers may be, it is impossible for any program to anticipate on the needs of each individual musician. Sooner or later you will need to do something that goes beyond the limits of every menu option. Customizing is the next step. Within Cakewalk, you can do that with the Cakewalk Application Language.[1]

Glen Cardenas created his tutorial – "The Secrets of Cakewalk's CAL Programming Feature" – in 1998. I discovered it on Internet. It helped me to understand some of the particulars of how the CAL programs interact with Cakewalk. However, I still missed – as 'old' computer programmer - good reference material about the language and how to use specific Cakewalk concepts within the programs.

Therefore, I decided to restructure the information, to add my specifics, and to update it for Cakewalk SONAR. At the end, I assume that it will be almost rewritten. However, in the current version readers, who know Glen's WEB-site, will still recognise parts of his tutorial. Glen Gardenas, who does not have time to update his WEB-site, gave me permission to publish this document. You may expect that over time this document will deviate more and more from Glen's tutorial.

Those of you, who have experience in computer programming, will recognize this document - more than Glen's tutorial - as a programming reference. I hope – due to my experience as musician – that it will also help non-programmers to understand the programming in the Cakewalk Application Language.

Ton Valkenburgh

[1] Text from Glen Gardenas, copied out of his "The Secrets of Cakewalk's CAL Programming Feature".

Prefaces

Preface with version 2 and 2.1

Based on the review of Frans Bergen, I have updated, and restructured the guide.

The main parts of the guide are:

- The CAL language (chapter 2);
- Creating the CAL program (chapter 3);
- Programming techniques (chapter4);
- Tips, and work-arounds (chapter 5).

I have added more information about the '**menu functions**'. I have not tested all functions with SONAR, and therefore cannot guarantee that the information is correct for SONAR. My experience has shown that especially in the '**menu functions**' there may be changes compared with previous Cakewalk version. This means there is still work to do in this area.

I like to thank Frans Bergen for reviewing this guide, and his help to improve the quality.

Preface with version 1.4

I have improved the explanation of the sendMIDI function and added an example in chapter 4: 'Programming techniques'. I added more routines to the library. The rest are editorial changes and some restructuring of the document.

Preface with version 1.3

Chapter 4: 'Programming techniques' has been expanded. In addition, new library programs are included which are used in tools of the MIDI-Kit.

Preface with version 1.2

I have added more SONAR specifics, library programs, and an index.

Preface with version 1.1

I have added some information about SONAR specifics. I will add more in later versions of the programming guide. I have added placeholders for more functions, but did not describe them yet explicitly. This version includes now descriptions of the example and library programs.

Preface with version 1.0

This version is almost an unchanged copy from the original WEB-site from Glen Gardenas.

1 An overview

Based on the LISP syntax the Cakewalk Application Language is not easy to understand for non-programmers – as most users of Cakewalk probably are. Also it has unfortunately many unexpected, and unnecessary limitations, which may reduce its usage.

In this programming guide the Cakewalk Application Language will be explained, and it can be used as a reference to the features of the language.

In addition a set useful routines are provided, which helps to overcome a couple of the limitations in the language.

Not every sequencer lets you - the user - create your own custom editing functions. Cakewalk has this feature. The Cakewalk Application Language makes it one of the most powerful sequencers. The programming feature allows you to expand Cakewalk to satisfy your needs. Just because you read the word '**program**' does not mean you need to be a certified programmer to create CAL programs. Although a background in computer programming will give you some advantage in learning CAL; all you really need to know, are a few basic concepts and you must have a clear idea of what you want to do with your MIDI-sequence. Therefore, it is important for the understanding of this guide that you already have a good understanding of the features of Cakewalk SONAR for Windows.

As far as CAL is concerned, what we will discuss here applies to Cakewalk SONAR. However, it may apply to all versions of Cakewalk from version 3 through version 9 too. Due to limitations in my time and the versions of Cakewalk in my possession, I only tested with Cakewalk SONAR 3. I will reference to many of Cakewalk's features and will spend little time explaining them. Much related information is in the user's book and the Cakewalk help files. Read these first, get used to what Cakewalk is already able to do, and then learn with this guide how to go beyond these built-in functions, and enter the world of CAL customizing.

This document focuses on CAL programs as written in SONAR, using functions arranged as program code. The information can also be used for earlier versions of Cakewalk. Specifics for earlier versions of Cakewalk can be found in chapter: 5: 'Tips, and work-arounds'.

1.1 The nature of computer programs

Before going into depth in the nature of writing computer programs we have to get a good feeling how programs will look like, and how they differ from what we are used to in 'normal' live. This is particular true for the Cakewalk Application Language.

It is not important to know how the sequence data is formatted and stored in a computer. You only need to know how it is presented to the programmer, and how Cakewalk uses the program statements to manipulate the data.

The best way to learn the art[2] of programming is by starting with a simple existing program and modifying it for your own purpose.

[2] Software programmers like to state that programming is an art. To my view, it is more a discipline.

2 The Cakewalk Application Language Fundamentals

CAL has data types, events, variables, constants and functions, which I will describe in this chapter. Not all of this is documented in the Cakewalk documentation. I will make explicit statements if the function, data type, or whatever is an undocumented one. Realize that using undocumented features can always give surprises, because in later versions of CAL they can disappear or behave differently.

2.1 The syntax of Cakewalk Application Language programs

A Cakewalk Application Language looks different from what you are used to. Therefore, we will first explain how the syntax of this program language looks like.

Let us first write down a normal calculation:

```
1 + 2
```

In CAL it will look like:

```
(+ 1 2)
```

Of course, both notations give the same result: **3.**

Normally you will not manipulate numbers, but data in your MIDI-sequence. This data is represented as variables or events.

So let us make our example a little bit more abstract:

```
(operator variable1 variable2)
```

The operator in our example is the **+.** CAL has many other operators. Later we will mention them all.

The syntax of CAL is like in LISP. Mathematical operations like "**addition**" and logical operations such as "**greater than**" are functions. That is first, you list the function, and then you list the arguments, or variables the function operates on. Keep in mind that an argument can be a variable, another function, or even a group of functions in parentheses. As humans, we say '**two plus three**' when we add. CAL says '**plus two three**' instead. Just remember, in CAL you must always put the type of function to be performed first followed by any arguments needed for the operation.

I will dig now further in the structure of the CAL programs. When writing in CAL and in most languages that use parentheses in this way, the program statements are within parentheses. The structure of a program is:

```
(do (op var var) (op var var))
```

Of course mostly programs are too big to fit on one line. Therefore, the program code tends to be arranged with one function on a line and the closing parenthesis of complex functions (functions made up with other functions) on their own lines. Indenting also makes the code readable. To demonstrate this, let us look at a logical program using filler words to represent functions.

```
(do                              ; each program starts with a do statement
   (declare a variable)          ; then you declare variables
   (declare another variable)
   (forEachEvent[3]              ; then the functions of the program
      (do
          (something)
```

[3] The "**forEachEvent**" function will be explained in chapter 2.4.1

```
            (another thing)
         ) ; end of do
      ) ; end of forEachEvent
   ) ; end of do
```

The "**do**" indicates that a sequence of functions will follow. Therefore, the first "**do**" indicates that the program will have several functions one after another to be run one at a time. These will be the first variable declaration, the second declaration and the "**forEachEvent**" function. I will handle the details of the "**forEachEvent**" function in chapter 2.4.1.

The reason for the declaration statements is, that we must make sure variables are declared before they are used by functions in the program code. In other words, if you are going to use a variable to hold the value of a note, say the highest note you wish some function to recognize, and you wish to call that variable, then you must tell CAL to expect to see it in the code and to save room for it in memory under its previously defined name. In this example, two variables are declared and then the "**forEachEvent**" function is executed.

The "**do**" after the "**forEachEvent**" statement lets CAL know that this loop is also made up of several functions.

You can see by the colors how the closing parentheses associate with the opening parentheses. As stated above you could also write the program like this:

```
(do (declare a variable) (declare another variable)(forEachEvent (do
(something) (another thing))))
```

Making a program readable is half the job. With proper formatting, you can match the closing parenthesis with the opening ones by the indention. Adding comments[4] to that will help even more, and ensure that later you still can understand your own program.

Keeping one function to a line also helps in maintaining a logical understanding of what the program is doing along its various points. Aside from making the code readable to humans, CAL could not care less if there is indention or if functions and closing parentheses are on their own lines. The form is for your own benefit. CAL does not care what the program code looks like, the program must only be written following the syntax rules.

I will go back to our original example that I made more abstract:

```
(operator variable1 variable2)
```

The operator in this example is the **+.** CAL has many other operators.

Of course, you want to store the result in another variable. Look at the following more complex statement:

```
(= result (+ variable1 variable2))
```

How does this work? CAL analyzes and processes first the information within the inner parentheses, and then the information within the outer parentheses. This is very important to remember for programming. The evaluation is form **inside** to **outside**.

Now we must assign a value to the variables, because otherwise we will not get any result. Depending on its use the size of a variable can vary. For numbers one of the used variable types is the integer. The program will now look like:

```
(int variable1 1)

(int variable2 2)

(= result (+ variable1 variable2))
```

[4] The texts behind the semi-colon (**;**) are comments, and ignored by CAL.

The problem with this program is that you have to change it if you want to calculate e.g. **3 + 5**. What you really want is that you do the calculation based on numbers, which are entered by the user. Therefore, let us request for user input for each number. In this example, I have added comments after the semicolon, which is the normal way of adding comments to a CAL program.

```
(int variable1 0)                              ;The default value of variable1
                                               ; is 0
(int variable2 0)                              ;The default value of variable2
                                               ; is 0
(int result 0)                                 ;The default value of the
                                               ; result is 0
(getInt variable1 "Number 1 = " -100 100)      ;Request input for variable 1
                                               ; with the text: Number 1 =
                                               ; values between - 100 and 100
                                               ; are allowed
(getInt variable2 "Number 2 =  -100 100)       ;Request input for variable2
(= result (+ variable1 variable2))             ;Add and store in result
(pause  "Result = " result)                    ;Display result and pause
```

I use in this program the function "**getInt**". This function prompts the user with a text, gets the input, and stores it in an integer variable. The other function I added is "**pause**". This function pauses processing, and prompts the user with text. The text can be built up by using contents of variables.

This is now almost a complete program. However, we have to add the start and begin indication of the program. All CAL programs are contained within a "**(do**" at the begin and a "**)**" at the end. Therefore, our program will look like:

```
(do
  (int variable1 0)                            ;The default value of variable1
                                               ; is 0
  (int variable2 0)                            ;The default value of variable2
                                               ; is 0
  (int result 0)                               ;The default value of the
                                               ; result is 0
  (getInt  variable1 "Number 1 = " -100 100);Request input for variable 1
                                               ; with the text Number 1 =
                                               ; values between - 100 and 100
                                               ; are allowed
  (getInt variable2 "Number 2 =  -100 100);Request input for variable2
  (= result (+ variable1 variable2))           ;Add and store in result
  (pause  "Result  =  " result)                ;Display result and pause
) ; end of do
```

You see also two conventions, which make reading the program easier. The statements within the do statement are indented, and to the closing parenthesis the comment: **; end of do** is added.

Comments are very important. The program does not use them, but they explain to the reader what the program is supposed to do. You should realise that, if you do not add enough clear comments to your programs, that after some time, you do not understand your own programs anymore.

Therefore, I show you now how the total program should look like. You will find the program in chapter 7 as 'Addition.cal':

```
;  Addition.cal
; © Copyright 2004 T. Valkenburgh
;  Version 1.0, April 2004
;
; This routine adds two numbers.
; Input parameters:      'Number 1 = ' (-100..100)
;                        'Number 2 = ' (-100..100)
;
(do
  (int variable1 0)                          ;The default value of variable1
                                             ; is 0
  (int variable2 0)                          ;The default value of variable2
                                             ; is 0
  (int result 0)                             ;The default value of the
                                             ; result is 0
  (getInt  variable1 "Number 1 = " -100 100);Request input for variable 1
                                             ; with the text Number 1 =
                                             ; values between - 100 and 100
                                             ; are allowed
  (getInt variable2 "Number 2 = "  -100 100);Request input for variable2
  (= result (+ variable1 variable2))         ;Add and store in result
  (pause  "Result  =  " result)              ;Display result and pause
) ; end of do
; end of program
```

To run this program in SONAR, you must select '**Run Cal**' from the '**Process**' menu, or from the '**Edit**' or '**Tools**' in other versions, and choose this program from the ones listed. Run this program, and see how the user interaction is. For using editors in detail see chapter 3.

This program has nothing to do with music. CAL programs will normally manipulate music events in a sequence of music event data. This example program is only created to show you the basic syntax, and structure of the Cakewalk Application Language. You must realize the nestled structure of the language. This structure does not always make it easy to read the programs, but remember the evaluation is from inside to outside.

2.2 Handling of event sequences by Cakewalk Application Language programs

It is important to realize that the sequences are organized by track and event times. If you ask an editor feature to operate on events from time X to time Y on tracks 1, 2, and 3, the software will start at time X and work on all of the events on track 1 until it reaches the last event at time T. It will then start over at time X on track 2 and do the same thing. It does this in one long pass, first one track and then the next. If you build a CAL program that must keep – in your mind - the events on each track separate, you can depend on CAL to operate only on the events in one track from the

starting time at the “From” marker to the end time at the “Thru” marker before going on to the next track. The values of the “From” and “Thru” markers are shown in Cakewalk panel.

Time in a sequence is different from the time you see on the screen. In a sequence, time is a raw number of clock ticks stored as a double word (32 bit value) from 0 to 4.29 billion. On the screen the raw time is displayed as ‘**measure**’, ‘**beat**’ and ‘**tick**’ or ‘**M:B:T**’ time. The raw time is converted to ‘**M:B:T**’ time according to the value stored in “TIMEBASE”[5]. For settings of 120 ticks per quarter note and a meter of $^4/_4$, there are 480 ticks per measure. CAL has functions that can convert the ‘**raw**’ time and ‘**M:B:T**’ time back and forth taking your settings into account.

CAL works on selected tracks, or part of selected tracks. To run CAL programs, you would set the “From” and “Thru” markers and highlight the track or tracks you wish to manipulate just as if you were about to use the ‘**Transpose**’ or ‘**Quantize**’ or any other feature in the ‘**Edit**’ menu. CAL programs will do nothing or unexpected things if no track or part of a track is selected. I recommend to add checking in your program whether a selection has been made. CAL does not provide special functions for that. Therefore, I will describe in chapter 4.2.1 ‘Checking the marking of Events’ how this can be done.

2.3 Data types, events, variables, and constants

Most of the data types, events and constants in the following pages will look familiar to anyone who has dabbled in CAL. You can also find brief descriptions in the Cakewalk help files. The truth of the matter is that there will likely be things here completely new to even veteran users because so much about CAL is undocumented. I have run into allot of the information in this tutorial by experimentation, luck and accident. Keep in mind that this compilation is based on my personal experience. That means that there may be more than I have documented here, and that I cannot always guarantee that this information base is absolutely complete. It is, however, a more complete source of information on CAL as I have encountered anywhere else, and so hope it will be of significant value to anyone who seeks enlightenment on the topic of CAL.

2.3.1 Data types

There are five data types in CAL and each is of somewhat different use. The main difference between the types is the size of number it can hold (aside from the “string” type, which is for text). Each can be assigned to a variable of your own making, but which is best for what kind of variable depends on the use to which it will be put. Here is a rundown of the five types listed by the abbreviation you will use when referencing them in a CAL program. Note that they must be spelled exactly as you see them here in order for CAL to recognize them.

You must be careful with mixing data types in calculations, because this can give unexpected results.

```
int
```

An “int” stands for a signed 16-bit integer and is the most common data type. It is a whole number ranging from negative 32676 to positive 32767. You will use the “int” data type for most of your flags, loop counters, temporary buffers, and just about anything that does not involve large values.

There is an important aspect. Assume you are going to scale values. You take a number, multiply it by some factor, and then divide it back down to a smaller number again. You must ensure that variable is big enough to hold the temporary results of such scaling even if the result would be well within the limits of an integer type.

[5] To set the number of ‘**ticks per quarter-note**’ You select from the ‘**Options**’ menu, the ‘**Project Options**’ heading, then the ‘**Clock**’ tab, and the ‘**ticks per quarter-note**’.

long

A “long” is a signed 32-bit integer that can range from roughly positive to negative 2 billion. Here is some room! This is a good variable to declare if you think you will be over-ranging an integer’s value during calculations.

word

A “word” is an unsigned 16-bit integer that goes from 0 to 65536. This whole number will always be positive. Note duration’s are stored as “word” values. Again, if you are going to be playing around with a duration value, you might want to declare a variable larger than this one just in case you send the value over the top at some point before you have a chance to correct it.

dword

A “dword” (double word) is an unsigned 32-bit integer that can range from 0 to over 4 billion. Event times are stored as double words. There is no variable type bigger, but none is likely to be needed in that 4 billion ticks at 120 ticks per quarter note would run almost 9 million measures. Even Wagner would have had a hard time writing a score that long!

string

A “string” is a series of ASCII characters between double quotes, or to put it simply, it is text. You can declare strings, build strings and send strings to some functions that use text, but unfortunately, you cannot input a string during the running of a CAL program. A string will look like:

```
"This is a string"
```

2.3.2 Events

CAL provides for our use several built-in variables and constants. The most important for everyday use are the constants and variables that correspond to the types of events in a sequence and the various parameters or data variables that belong to each one. You should memorize the following list if you plan to do a lot of work in CAL. Again I must stress that you must use exactly the same spelling, capitalization and punctuation format as you see here in order to use them. Note that many CAL internal variables have a period between words, use capital letters in the middle of names and other odd things.
Normally you refer to variables and constants by name. However, they have an internal number which I also present.

All events have parameters kind, time, and channel. All event parameters are stored in variables.

Event.Chan

CAL puts that MIDI channel number in this variable when an event is being scanned by the “**forEachEvent**” loop. There are two things to remember about event channel assignments. First, this channel number will have no effect upon what channel the event will be sent over if the track in which it resides has a channel assignment. The track channel will always override the event channel. If the track has no assigned channel, then the event channel determines the final channel it will be sent over. Second, even though we number the MIDI channels from 1 through 16, Cakewalk internally numbers them one digit lower from 0 to 15. Therefore, in a CAL program you must use 0 to 15 for channels.

As a side note, the same is true for track number assignments. We see tracks 1 through 256 and CAL sees them as 0 through 255.

Event.Kind

CAL manages this variable and you should not attempt to assign a value to it. CAL gives it a value based on the specific type of event in the sequence currently being accessed during a "**forEachEvent**" loop. The specific number this variable holds, is of little importance. Usually it is not necessary to view it or calculate with it. Just for the record, the number it holds will be 144 if the event is a note, 176 if the event is a controller, 224 if the event is a pitch wheel and so on. The reason you don't need to know this (except for downwards compatibility issues as noted above) is that there are constants in CAL (see below) that equal those numbers and all we need to do is see if "Event.Kind" is equal to one or more of them. Note that the two words "**Event**" and "**Kind**" have a period and no spaces between them. This will be the case with allot of these CAL variables, so pay attention to this detail.

Event kinds are:

"CHANAFT" (Channel after touch)	"NOTE"
"CHORD"	"NRPN" (Non-registered Parameter Number)
"CONTROL"	"PATCH"
"EXPRESSION"	"RPN" (Registered Parameter Number
"HAIRPIN"	("SYSX", "SYSXDATA" (System Exclusive data message)
"KEYAFT" (Key after touch)	"TEXT"
"LYRIC"	"WAVE" (Audio)
"MCI" (Windows Media Control Interface command)	"WHEEL"

The constants: "CHORD", "EXPRESSION", "HAIRPIN", "LYRIC", "MCI", "NRPN", "RPN", "SYSX", "SYSXDATA", "TEXT", "WAVE" allow an "Event.Kind" test to locate these "Events" in a sequence, and yet there are no provisions for viewing, accessing or changing the values associated with them. This makes their use rather limited to merely hunting for this type of event, but not being able to do anything with it, once it has been found, except deleting them! Nevertheless, I list them here mostly because, for many of them, their existence as they relate to CAL is undocumented. These constants are only available in the versions of Cakewalk that support these data types as part of their feature package.

CHANAFT

This "**Event**" constant is equal to 208 and is the "**Channel After touch**" event. Not all synthesizers support it. "CHANAFT" has only the variable: value.

ChanAft.Val

The one variable associated with a "**Channel Aftertouch**" event holds the strength of the after touch. It is an "int" and can be from 0 to 127.

CHORD

Chord names can be added to a sequence to assist guitar players following along with the music. These events are entered in the "STAFF" View. The constant equals 7. CAL programs cannot access the variables.

CONTROL

This is the "**Event**" that holds the constant value 176 for a controller event. Like "NOTE", we use this constant, among other things, to test "Event.Kind" to see if the current event is a controller.

"CONTROL" will have the variables number and value.

Control.Num

CAL sets this variable with the controller number value for events that are of the "CONTROL" kind. For example, a value of 7 corresponds to a volume controller event and 10 to a pan controller event. Changing this value changes the type of controller we are working with. This variable is of the "int" type and can range from 0 to 127.

Control.Val

The value or strength of a controller event is placed in this variable. It is an "int" and ranges from 0 to 127. Changing this value changes the amount of controller action for this event.

EXPRESSION

The expression event equals 5, and can be inserted into a sequence in the '**Staff**' view. They guide a musician in the performance of a piece. The event does not affect the MIDI performance; it is a staff view event only (be aware that it is not the same as MIDI controller 11). CAL cannot access its variables.

HAIRPIN

These events correspond to the Diminuendo and Crescendo marks available in the '**Staff**' view. Their constant equals 6. The event does not affect the MIDI performance; it is a staff view event only. CAL cannot access its variables.

KEYAFT

The "KEYAFT" ('**key aftertouch**') event is denoted by this constant of 160. Some synthesizers do not recognize this event, and still others will receive it but not send it. "KEYAFT" has the variables: key, and value.

KeyAft.Key

CAL fills in this variable with the pitch number when it sees a "KEYAFT" event. Like "Note.Key", it is an "int" from 0 to 127 and denotes the pitch or in this case, the physical key number associated with the event.

KeyAft.Val

CAL gives us this value when the "Event.Kind" is a "KEYAFT". Like the "Note.Vel" variable, it is of type "int" and denotes the strength of the '**aftertouch**' in values from 0 to 127.

LYRIC

This constant equals 2. The data associated with this event is a single syllable or word that is to be sung at that instant in the song. These lyrics can be displayed during playback in the '**Staff**' view. The variable contains a single text word of syllable, and is not accessible by CAL.

MCI

An MCI command tells Windows to do something with one of its multimedia devices at that instant in the song. See the user manual or the Windows Multimedia Programmers' Reference for more details. Its constant value is 4. The variable contains the MCI command text. CAL cannot access its variables.

NOTE

This is one of the events that allow us to test "Event.Kind" and also to assign the event within an "**insert**" function. Simply put, "NOTE" always equals the number 144, which is the CAL value for a note event.

A "NOTE" will have the variables: key, velocity, and duration.

Note.Key

This variable is one of the parameters of a "NOTE" event. It contains the pitch number of the note. It is of type "int" or integer and can range in value from 0 to 127. CAL fills in this variable with the pitch value whenever "Event.Kind" equals "NOTE", and you are free to work with, calculate with or even change this value if you want to alter the pitch of the note event.

Note.Vel

Here is another parameter of a "NOTE" event. It is of type "int" and contains the velocity of the note. It can range in value from 0 to 127. Like "Note.Key", CAL fills in the value for us whenever the current event is a "NOTE" and we are free to use and change it as we see fit in order to change the note's velocity.

Note.Dur

The duration in ticks of the note event is available to us through this variable. It is of type "word", can range from 0 to 65535 and is provided by CAL when the "Event.Kind" equals "NOTE". Changing this value moves the MIDI "**Note Off**" message for that note back and forth in time relative to the "**Note On**" message.

NRPN

The "NRPN" or "**Non-Registered Parameter Number**", which is 10, are clusters of four controller messages which Cakewalk versions 6 and above, and many synthesizers read as single messages. Earlier Cakewalk versions can send and receive them, but they appear as clusters of controller messages. Each "NRPN" event contains two numbers, a parameter value and a data value, each being a number between 0 and 16383. CAL cannot access the variables.

If access to such controller events is important to you, such as the example of accessing pitch bend depth, then see the chapter 5.9 'Details on RPN and NRPN Controller Events' for a way around this problem.

PATCH

This event equals 192 and denotes a patch change. "PATCH" will have the variables: number, and bank.

Patch.Num

As you might guess, this is the patch number associated with the "PATCH" event. It is an "int" from 0 to 127.

Patch.Bank

The instrument bank can be found or changed in a "PATCH" event as well, and this variable contains the bank number. It can range from -1 to 16383 with -1 meaning "**undefined**".

RPN

The "RPN" or "**Registered Parameter Number**", which is 9, are clusters of four controller messages which Cakewalk versions 6 and above, and many synthesizers read as single messages. Earlier Cakewalk versions can send and receive them, but they appear as clusters of controller messages. Each "RPN" event contains two numbers, a parameter value and a data value, each being a number between 0 and 16383. CAL cannot access the variables.

If access to such controller events is important to you, such as the example of accessing pitch bend depth, then see the chapter 5.9 'Details on RPN and NRPN Controller Events' for a way around this problem.

SYSX

This constant equals 0 and is for a "**System Exclusive Bank**" message. Its existence is documented as a CAL constant even though it is impossible to access the data associated with the event. The variable contains the Sysx bank number (0-255). The bank actually holds the data. The variable is not accessible by CAL.

SYSXDATA

System Exclusive data is MIDI's way of letting each synthesizer manufacturer transmit private data about their products. The constant value is 8. A System Exclusive message has a manufacturer ID; the rest of the message is completely proprietary and varies for each manufacturer, even for each of their products.

"SYSXDATA" has the actual data bytes to be sent as part of the event data. Again, none of this data is accessible.

TEXT

Text events equal 1. They are sort of like notes you write to yourself and place in the sequence to serve as documentation or reminders. TEXT will not be visible in the Staff view. CAL cannot detect TEXT events. When a test is done on a TEXT Event, CAL will do a test on a "LYRIC" event.

The variable contains text, but is also not accessible by CAL.

WAVE

This value of 3 corresponds to the "**Audio**" events that can be added to a sequence. The variables are: name, velocity, and number of samples. The variables are not accessible by CAL.

WHEEL

This constant identifies a pitch wheel event. It equals 224. "WHEEL" will have the variable: value.

Wheel.Val

Here we have a variable that can be positive or negative. It is of type "int" and ranges from 8191 to -8192. When we have a "WHEEL" event, CAL fills in this variable with the pitch bend amount. We are free to change it as we see fit, but remember, be prepared to deal with either a positive or negative value.

Event.Time

CAL fills it with the "**raw**" time of whatever event is being evaluated by the "**forEachEvent**" loop. It can also be assigned a value to change the time of an event or w*hen inserting events. It is of type "dword" and so can be from 0 to over 4 billion.

2.3.3 Marker variables

From

This variable is the same as the "From" marker in the Cakewalk front panel. It is the starting point for the **selected** part of a sequence. This value can be changed during the running of a CAL program any place **other than** within a "**forEachEvent**" loop. This makes sense in that the "**forEachEvent**" loop operates on events starting at the time stored in "From" and so cannot alter its own starting point. "From" is of type "dword" and so can range from 0 to over 4 billion. If, during a CAL program, you change the value stored in "From" before a "**forEachEvent**" loop, the loop will start at this new location in the sequence. Thus, you can make the assigning of this variable, and so the starting point of the loop, subject to conditions or calculation results within the program itself. This can be very handy! Keep in mind, however, that just as Cakewalk will not allow you to assign "From" a number larger than the value stored in "Thru", neither will CAL. The offending value will be automatically reassigned to correct the condition.

Now

This variable corresponds to the position of the bar cursor in the sequence and holds the value visible in the "Now" time window on the screen. It is the place where playback would start if the '**Play**' button were clicked on the front panel. It has little use in CAL. There are some strange limitations to the size of numbers this marker variable can hold. Even though it is 32 bits wide, assigning it a number close to that limit can crash some Cakewalk versions earlier than 7. See the chapter 5.6.2 'Hidden traps in the CAL editor' for more information.

Thru

"Thru" is the same as "From" except, of course, being the end marker of the selected part of a sequence. Everything said above about "From" applies to "Thru".

End

The final marker variable is really a '**read-only**' variable. It holds the location of the end of the sequence and thus cannot be changed from within a CAL program.

2.3.4 Constants

As a useful convention, CAL constants are named with all capital letters so they are easy to spot.

FALSE

This is a boolean constant is the other side of the "TRUE"/"FALSE" coin. It holds a value of '**0**'. See the definition for "TRUE" below for details.

TIMEBASE

This constant provides the number of ticks per quarter note the user has selected for keeping time in Cakewalk. The default value Cakewalk provides when it is installed, is depending on the version of Cakewalk/SONAR, but can be changed by the user.[6] Because CAL sees time as the number of ticks

[6] In certain (sub) versions of SONAR 3 the "TIMEBASE" constant does not vary with the user setting, and will always be 960. This means that SONAR will work with a wrongly calculated raw time. The bypass for this bug is to set the ticks per quarter notes on 960. This will then become the new default.

from the start of the sequence (referred to as '**raw**' time) and humans see it as the number of measures, beats and ticks in the composition (referred to as '**M:B:T**' time), this value is an important factor in equations that must take both time keeping systems into account.

This '**constant**' is a variable from the user point of view; because it can be set for each project. CAL cannot change "TIMEBASE**"**.

TRUE

This is a boolean constant that holds a value of '**1**'. This constant is used in functions that require a Boolean value of **yes** or **no** in order to signify the selecting or de-selecting of some parameter. For example, if you are setting up an "**EditCopy**" function, there are arguments that correspond to the check boxes used to set up the '**Edit**' menu '**Copy**' command. In order to translate to CAL the act of checking or unchecking these boxes, the corresponding variables will be given a value of '**1**' for checked or '**0**' for unchecked. It is allowable to substitute the constants "TRUE" and "FALSE" for '**1**' and '**0**' in these arguments.

VERSION

This constant contains the version of CAL currently running on the user's computer. It's **undocumented**, but useful as an error checker. A test should be made at the start of every CAL program to make sure the user has the proper version of CAL to run the program. This is most conveniently done using the "**include**" function to run a small CAL program called '**Need20.CAL**' from within any larger program to check this "VERSION" value and abort if it shows that an incompatible version of CAL is running. If you are creating CAL programs that can only run on higher versions of CAL, you can use the in the 'Library' (chapter 8) provided include program '+need version.cal' which test the CAL level based on input from the calling program.

The version number consists of a version and sub-version, divided by a dot. CAL does not use the dot. This means that e.g. a version 2.0 is seen in CAL as 20.

2.4 Functions

Without the ability to alter the values in variables, a program would not be of much use. It is with this in mind that we now introduce the various ways that we can manipulate the variables listed in the previous chapters. Remember, all of the **operations** you will be looking at in this section are real functions. That means that the way the elements of the operation are ordered will follow the rules of function syntax and as such probably look a bit weird to you at first. Just keep in mind that as with all functions, the function name (or in some cases, symbol) comes first followed **in proper order** by the arguments that will be operated on.

In the examples, I will sometimes use not yet described items, but I still believe that the examples help to make things clear. We will start with a very important function in CAL. It is key to understand this function, because it is one of the basics of CAL operation.

2.4.1 (forEachEvent) <function>)

As already stated earlier, CAL programs mostly will manipulate selected events in the selected tracks. The defined actions within the function "**forEachEvent**" apply to the selected Events. In a program it will look like:

```
(forEachEvent (function))
```

or more like you are used to seeing:

With the sample program 'Timebase.cal', you can check whether your version of Cakewalk has this bug.

```
(forEachEvent
    (function)
)
```

As already mentioned, this function is the major workhorse of CAL. It takes the events in the selected part of the sequence (the area between "From" and "Thru" on the tracks you have highlighted) and runs each event through it has nested functions, one event at a time and in sequential order, until all of the events have been scanned. It then aborts and CAL continues with the next function after the loop's closing parenthesis. You can run as many "**forEachEvent**" loops as you want in a CAL program so long as you don't try to run one inside of another.

The "**forEachEvent**" function can have only one function in its nest. In order to get things done, that function is often a compound function made up of nested functions. This compound function is usually a "**do**" or an "**if**" function with lots of nested functions like so:

```
(forEachEvent
    (do
        (somefunction)
        (anotherfunction)
        (morefunctions)
    )
)
```

The "**do**" tells CAL that even though it would expect only one function for this loop, there will be several instead. This is because the three functions are not nested together. Each stands alone. We therefore use "**do**" to create a nest for them all to share so that the "**forEachEvent**" loop still sees just one function. The function "**do**" will be explained in the chapter 2.4.10: 'Control flow functions'.

When this loop is running, CAL places the kind of event currently being serviced in the variable "Event.Kind", the event's starting time in "Event.Time" and MIDI channel in "Event.Chan". The event's parameters are placed in the other event variables as applies. Notes will have their parameters available in "Note.Key", "Note.Vel" and "Note.Dur" and so on for each type. Also during the running of this loop, the marker variables "From", "Thru" and "Now" are no longer responsive to any changes you might try to make to them.

During this processing CAL keeps a copy of the selected events. If you insert a new event at an "Event.Time" higher than the spot the loop is currently at, CAL will not see the new event when it comes to the new event's time until the loop is over. If you were to run a second "**forEachEvent**" loop, the events inserted by the last loop would be available to this new loop.

2.4.2 Declaration Statements

CAL allows you to initialize variables, but does not allow you to define constants. If you want to use a constant in your program you have e.g. to declare an integer with the required value.

Note: Before using a variable in a CAL program, you must first have declared this variable.

(int <name> [<value>]) [7]

This declares an integer that you can name whatever you want. You should use names that help remind you of the variable purpose such as '**highnote**' or '**counter**'. After the name comes the optional value, it will be initialized to. If nothing else, initialize it to zero. Realize that if the variable does not get his initial value, that the value cannot be predicted.

You can also initialize a variable to the value of another variable so long as the latter has been declared and initialized. Suppose you declare a variable like

```
(int range 12)
```

and you want to declare another variable later that equals 'range'. Also, suppose that in the meantime, '**range**' has been subjected to other functions and so it is value is unknown. You could do it like this:

```
(int lowpoint range)
```

and now '**lowpoint**' will be initialized to whatever value currently resides in 'range'. You could also have declared '**lowpoint**' and initialized it to zero at the same time you declare "range" and then when the time comes just set '**lowpoint**' equal to 'range' like

```
(= lowpoint range)
```

which is the usual course of action. If you are one of those efficiency freaks that must use the absolute minimum number of statements to perform a task, you can save yourself the extra '**equal**' function by declaring at the time of assigning the value instead. Another way to save a few bytes is to chain the declaration of like variables. This means that you can list several variable names on the line after the "int" keyword, each separated ('delineated' in geek speak) by a space, and then at the end given an initial value. All of them will be initialized to that same value. Such a statement might look like this:

```
(int first second last loop lownote highnote maxvel minvel temp 0)
```

In this statement, all of the variables listed from '**first**' through '**temp**' would be declared as integers and initialized to zero. I don't like to do this because as luck would have it, at some point I might need to change the initial value of a variable as I'm shaking down my program and would have to edit this statement to remove the offending variable and then write a new one just for it. I like to keep my variables separate so all I have to do to change them is alter one statement. It also makes them easier to read and keep track of. Remember that an "int" has a range of -32676 to 32767 so be sure all of your integer variables will stay in that range throughout the running of your program.

(dword <name> [<value>])

This declares a double word called '**name**' initialized to a value of '**value**'. These variables are positive only and can range from 0 to 4,294,967,295. Everything said about "int" applies here as well.

(long <name> [<value>])

This declares a long variable. It is able to swing plus or minus 2.14 billion. Again, the rules are the same as noted above.

---

[7] <name> indicates a required parameter name;
[<value>] indicates an optional parameter value;
parameters can be constants, variables, or functions.
See also chapter 10: 'Document conventions' 'Backus Nauer Form'.

```
(string  <name> ["This is text"])
```

Here we have a variable declared as a string called '**name**' and initialized to the text enclosed in *double quotes*. Strings can be very useful when we want our CAL program to generate track names or some such. Like its kind above, you can chain declare multiple string variables and initialize them all to the text in quotes, but I see little use for such an occurrence. There is a limit to the size of a string variable, 126 characters. Rarely would a string of more that two-dozen characters be needed, so that issue will probably never come up in general use.

```
(undef  <name>)
```

This function **undefines** or erases from memory the pocket occupied by a variable and deletes its name from the list of declared variables. It works on any variable you have declared using any of the above data types. This is a way of freeing up memory for new variables after a current one has outlived its usefulness. It can come in handy for erasing large variables like double words to make room for more integers for example. From time to time, I have run into the limits of CAL's ability to keep up with too many variables. The use I tend to put this function to the most is undefining variables that an included program declares, but does not need to pass back to the parent program. See chapter 2.4.10 for an explanation of the "**include**" function.

```
(word  <name>  [<value>])
```

This statement declares a word variable named '**name**' with an initial '**value**'. Everything mentioned above applies here. Its range is limited to positive numbers and can go from 0 to 65536.

2.4.3 Assignment Functions

The following are mathematical assignment functions. They start with the most basic, the 'equals' function, but then slide into a gray area between assignment and mathematical. I will list them here before the mathematical functions they emulate because they do possess assignment characteristics.

```
(=  <variable>  <value>)
```

This "**equal**" function assigns the value of '**value**' to the '**variable**'. Take a close look at the sentence you just read! It is '**variable**' that will be assigned and '**value**' that holds the master value that will be passed on by the "**equal**" function, not the other way around. In English, it would read '**Store the value into the variable**'. Do not get confused on the direction of flow in this and all following mathematical functions, as it is easy to do given the LISP syntax of CAL. Not surprisingly, that '**value**' can be a variable like "From", or even another function. For example, you could have something like:

```
(= lowrange (- Note.Key base))
```

Here, the variable '**lowrange**' is set equal to the result of subtracting '**base**' from "Note.Key". Notice the parentheses. The function "**minus**" with its two arguments must be enclosed by its own set of parentheses, but this does not lessen the necessity of enclosing the "**equal**" function with a complete set of parentheses as well.

```
(+=  <variable>  <value>)
```

This assignment function acts as a form of shorthand for making a variable equal to itself as modified by a value. In English, this would read '**Add the value to the variable and store the result into the variable**'. As with any equation, '**variable**' can be a constant or another function. In CAL longhand, it would read:

```
(= variable (+ variable value))
```

By the way, the double use of '**variable**' in this longhand example is legal. You can use the same variable as both destination and source because CAL first does the calculations and then passes the result to the destination variable. This way, the contents of the variable are safely preserved until after it is used in the calculation, and only then, it is changed to its new value. The "+=" function is simply a bit more efficient.

```
(-= <variable>  <value>)

(*=  <variable>  <value>)

(/=  <variable>  <value>)
```

These are just like the first example ("+=") except they represent subtraction, multiplication and division respectively.

```
(%=  <variable>  <value>)
```

This one may take some explaining. The percent sign means '**Find the modulo value**'. A modulo value is the remainder left over after dividing the first variable by the second (value). CAL does not see fractions, fractional amounts or decimal values. It deals strictly in whole numbers. Therefore, if you divide 9 by 8, you will get an answer of 1. You will get the same result for dividing 9 by 7, 9 by 6, or 9 by 5. The result of dividing 9 by 4 would be 2. The remainder left over by any of these divisions is only available by obtaining the modulo value. The modulo value from dividing 9 by 8 is 1, as that is the remainder from dividing them. For 9 by 7 the modulo value would be 2, for 9 by 6 it would be 3 and so on. We can use this value after performing a "**divide**" function for various purposes such as rounding. Using the modulo assignment function is a bit more obscure than the rest of our mathematical assignments described above. We are more likely to use a modulo value in the conventional mathematical functions.

2.4.4 Input functions

Once we declare a variable, it is a fair game! However, there will be many times when the operation of a CAL program will depend on the ability of the user to set some of the parameters. Thus, we come to the "**get**" statements. There is a "**get**" statement for the integer and word and a special way of entering a double word, as we will see shortly. There is no input statement for the string unfortunately; it would be nice to input strings from time to time. However, alas, we have not the luxury. As far as the numerical variables go, we are free to query the user for input at any time using the following statements:

```
(getInt  <name> <prompt> <minimum> <maximum>)

(getWord <name> <prompt> <minimum> <maximum>)
```

As should be obvious, each statement is meant for a specific data type. Note how the "**get**" part is in lower case and then without any space comes the name of the data type with its first letter capitalized. Next comes the name of the variable we are giving the user the opportunity to change. Then comes a "string" that will serve as the prompt for the user. It can be anything you wish to convey, just limit it to less that a screen's width because the resulting dialog box will spill off the screen. Last, come the restrictions you can place on the user's response in the form of two values. This is optional but recommended. The first is the smallest, or in the case of an "int", the lowest value you will allow. The second is the highest value allowed. As I mentioned earlier in this document, I like to leave a space or two at the end of the prompt before closing the quotes. The reason is that the dialog box displays a small window after the prompt containing the current value of the variable and a blinking cursor. The spaces allow a bit of esthetic room between the text and that little window. The user can just hit '**Enter**' or click '**OK**' and accept the current value or can enter a new value and then hit '**Enter**' or click '**OK**'. If they attempt to enter a value outside the limits you have set with '**minimum**' and '**maximum**', an error box appears displaying those limits and allows the user to try again until an acceptable value is entered. Your choice of '**minimum**' and '**maximum**' will depend on the nature of the variable. If you are requesting a MIDI channel number, the limits should be 1 and 16. If you are requesting a pitch, you would use 0 and 127 as your limits and so on.

```
(getTime <name> <prompt>)
```

This is a bit different in that the user is to enter a time value in '**measure**', '**beat**' and '**tick**' that CAL will convert into '**raw**' time and assign to the double word variable '**name**'. This variable **must** be of type "dword" and already been declared. Keep in mind that you can have a '**tick**' value of **0**, but '**beat**' and '**measure**' must be a value of at least **1**. Also, if you enter a beat that does not exist, such as 6 in 4/4 time, or a value of '**tick**' greater than the value set in "TIMEBASE", CAL will convert the values anyway by multiplying the number of '**beat**' by "TIMEBASE" and adding to the result the value in '**tick**'. If you reconstruct the '**raw**' time back into '**M:B:T**' time (see "**makeTime**" in chapter 2.4.11: 'Musical time functions'), all of the values will be corrected to display a proper time given the '**meter**' and "TIMEBASE" values in use during that Cakewalk session.[8] *The other difference is that CAL will not allow the use of* ***'minimum'*** *and* ***'maximum'*** *limits for "***getTime***". Attempting to use limits, results in a* ***'wrong number of arguments'*** *error at run time.*

2.4.5 Output functions

CAL gives us the chance to send information and data to the user and to the MIDI port. Along with these output functions we will also discuss the "**format**" function, which converts variable data into text for e.g. use within functions which places text into the '**Track**' view screen fields (and thus into the sequence file).

```
(message <operand1> [[<operand2>] ……])
```

This function places a message on the status bar at the bottom of the screen. It does not interrupt the operation of the program, just displays the message. The "**message**" operands can be strings (text between double quotes), integers ("int", "word", "dword"), or a function. The "**message**" function

[8] This feature has a bug in it for CAL versions 7 and 8. The ("**getTime**") dialog, which is supposed to request user input in '**M:B:T**' time, instead will only accept '**raw**' time.

will display the operands in the form of numbers or any mixture of the two. There is an important thing to consider about sending messages to the status bar. While a "**forEachEvent**" loop is running, CAL has exclusive access to the status bar. However, outside this loop, anything CAL puts there can be overwritten by a Cakewalk internal function message like letting you know that an '**Autosave**' is going on. '**Autosave**' can happen during CAL programs. For that reason, you cannot be assured the message will be there long enough for the user to read it. Needless to say, after the CAL program ends and supposing there is no '**Autosave**' processing, just pointing the mouse at a display field will replace the message with the description of what the cursor is pointing at. If the pointer happens to be over such a spot when CAL stops, any message will be immediately overwritten. This aside, the message is a good way to let the user know where the program is and that it is alive and running, less the user panic during a long loop and think the system has locked up. There is a price to pay for this function: time. It takes longer to run a CAL program if there is constant updating to the status bar. Consider this code fragment:

```
(forEachEvent
     (do
         (message "Processing Event # " (index))
         (if  (so on and so on.....
```

CAL will be sending a new message to the screen every time the "**forEachEvent**" starts a new pass. While this can be a great way of keeping the user transfixed on the status bar as numbers flicker by at a rate too fast to read, it also slows down the system a little. You must weigh the costs and benefits of providing such an ongoing tally message. Remember to leave a space between closing quotes and variables because unlike some versions of BASIC, CAL does not format the display of variables and text for you.

```
(pause <operand1> [[<operand2>].....])
```

This is similar to the "message" function and again you can combine strings, integers ("int", "word", "dword"), or functions. The difference is that "pause" halts execution of the CAL program and displays a dialog box in the middle of the screen containing the information the user provides in the function statement plus '**OK**' and '**Cancel**' buttons. The user must click '**OK**' or hit the '**Enter**' key to acknowledge the "**pause**" box before the program will continue. The user can also click the '**Cancel**' button or hit the '**Esc**' key and abort the CAL program completely.[9]

Besides displaying information critical to the user's decision to continue the program or cancel, the "**pause**" is a great troubleshooting tool for programmers. If a CAL program is not acting quite right or if you want to verify the operation of some part of a program, you can place "pause" statements at key locations in the code to display the contents of some important variables and see if things are running as planned. Note this example:

```
(if (> something somevalue)
     (do
         (pause "Reached first test.  somevalue = " somevalue)
         (go on about your business.....
```

Here a "**pause**" has been placed at the '**then**' part of an "**if**" function to help verify if that function is being called. If it is, the "**pause**" tells the programmer where the program currently is and what the current value of variable '**somevalue**' is. After the programmer hits '**Enter**' or clicks '**Ok**', the program goes on as if nothing ever happened. By the way, there is a limit of 128 characters for a "**pause**" message, so if you have allot of information to give out, it may require more than one "**pause**" box to deliver it.

[9] By the way, hitting the '**Esc**' key at any time in the running of a CAL program will cause it to abort and exit.

(format <operand1> [[<operand2>]…])

Like the "**message**" and "**pause**" functions above, "**format**" takes any combination of strings and integers values ("int", "word", "dword"), or functions and generates a string that can be passed to another function or assigned to a "string" variable. Take this example:

```
(int track 0)
(string name "No Note Has Been Found")
(forEachEvent
     (if (== Event.Kind NOTE)
            (do
                (= name (format "Note Number " Note.Key)
                (TrackName name track)
                (++ track)
            )
     )
)
```

What we have done here is declare an integer variable that we will use for track numbers and initialize it to 0 so it points to the first track[10]. Next, we declare a string variable and initialize it to a default text string. Next, we enter a "**forEachEvent**" loop and our "**if**" function filters out anything except notes. For each note we find, we use format to convert "Note.Key" into a number and combine it with the text string in quotes. The result is assigned to the string variable 'name' and using the "**TrackName**" function, we place this text in the on-screen '**Name**' field for track number '**track**'. Now we increment '**track**' so that upon the next occurrence of a note, it will point to the next track and we loop again. When the program stops, aside whatever else the program containing this fragment does, it will have named tracks for each note in the selected part of the sequence. If you wanted to eliminate the '**track**' variable all together, you could replace the first two statements after the "**if**" with this single statement:

```
(TrackName (format "Note Number " Note.Key) track)
```

As a part of this program, you would also likely end up cutting out all of the notes of each pitch and pasting them into the appropriately named tracks. This is useful with percussion tracks where it's convenient to separate each percussion instrument onto its own track so it can be manipulated independently.

(sendMIDI <port> <channel> <kind> <data1> [[<data2>]….])[11]

You can send a MIDI event during a CAL program with this function.

Use '**port**' as the port number (0 .. 15) you want to send over. If '**port**' is set to -1, the function will send the message through all ports. The '**channel**' variable is the channel number (0 .. 15) and -1 will set the function to send over all channels. The '**kind**' constant is one of the recognized event types such as "NOTE" or "CONTROL" just like with the "**insert**" function. Use '**data1**' and '**data2**' for the information to be sent. Not all event kinds use both data bytes, and you must use the proper formatting of the data bytes for the event in question. '**data2**' is only valid for "NOTE", "CONTROL" , and "KEYAFT". Remember, MIDI data bytes must be between 0 and 127 except for type "SYSX" (system exclusive message).

---

[10] Remember, we see tracks 1 through 256 and CAL sees them as tracks 0 through 255.

[11] This function only works for CAL versions above 3.1. The version number 3.1 is presented as 31 within CAL.

The "SYSX" message is formatted by sending the number 240 (F0) before '**data1**', and after sending the number 247 (F7) after your last data.[12] Check your synthesiser MIDI implementation specifications for details on how to use system exclusive messages with your instrument.

The value of <channel> will be ignored for "SYSX" messages.

2.4.6 Buffer functions

The buffer functions work on the sequence itself. They are "**insert**", "**delete**" and "**index**". These functions add, subtract and display the position of an event in the sequence. This will make sense as we look at them.

```
(index)
```

This function returns the number of the event being scanned by a "**forEachEvent**" loop as it is ordered within a track sequence (buffer). The very first event in a track sequence is numbered 0 and the next 1, then 2 and so on. If you delete an event, all events after it are moved back in number so that the event following the deleted one now has its number. Inserting an event causes all events after it to move up in number to make room and so on. I have seen only two applications for "**index**". One is in a "**message**" function to tell the user how many events have been scanned, the other is in an "if" function to let the program know that it has reached the first event on a new track by testing for (= = (index) 0).

```
(insert <time> <channel> <kind> […data parameters…])
```

In this function we use the keyword "**insert**" followed by the raw time where we want the event inserted into the sequence. This is followed by the MIDI channel number we wish the event to be assigned and then the kind of event we wish to insert. The '**data**' variables correspond to the parameters for that kind of event. Not all event types will need all three '**data**' variables. Here is a rundown of specific "**insert**" functions for each event type:

```
(insert time channel NOTE key velocity duration)
(insert time channel CONTROL number value)
(insert time channel WHEEL value)
(insert time channel PATCH number bank) ;The 'bank' variable is optional
(insert time channel KEYAFT key value)
(insert time channel CHANAFT value)
```

As you might guess, the keyword "**insert**" and the constant for the event type such as "NOTE" and "WHEEL" must appear in your function as shown, but all of the other parts can be variables or functions as you require. If you perform an "**insert**" in a "**forEachEvent**" loop, the event is inserted into whatever track is currently being scanned.[13] If you have more than one track highlighted, you may have a hard time knowing when the loop has ended one track and gone on to the next one. Then again, it may not matter providing the insert is keyed to the events around it. If the insert is not within a "**forEachEvent**" loop, it will be inserted into the track highlighted by the cursor[14] regardless of whether or not there are other tracks selected or where the tracks fall numerically. The '**bank**' number in the '**Patch**' "**insert**" function is optional. If it is not supplied, it is assumed to be the same as assigning it -1, which means '**none**' and shows up as '**---**' to the user in the '**Track**' view.

---

[12] Make sure that you do **not** include these "SYSX" message start byte F0 and end byte F7, in your data, because CAL will supply them. See also the program "GM Mode.cal" for an implementation.

[13] The recommended approach is to use "**insert**" within a "**forEachEvent**" loop.

[14] See also chapter 5.5: 'Explicit and implicit track selection'.

```
(delete)
```

This function is only usable within a "**forEachEvent**" loop and will delete from the sequence the event currently being scanned by the loop. It has no arguments or data associated with it. He is just the hatchet man and he works alone. Once an event is deleted, it is gone forever. If you first store all the information about the event before you delete it, it can then be reinserted later using the "**insert**" function. These two make a good team when writing programs that must store an event and then act upon it based on information gained from another event during another pass of the "**forEachEvent**" loop. Remember, the events scanned by that loop are only available for manipulation during that pass of the loop. Once the loop starts on another pass, the first event is out of reach and a new event becomes the focus of the loop. In order to change an event after the loop has passed it, you must store the event in variables and then delete it while the loop is scanning it. Later, once you have whatever information you need to alter the event, you can then reinsert it back into the sequence at its original event time regardless of where the loop currently is.

2.4.7 Mathematical functions

Let us move right into the mathematical functions. They are very straight forward, and by now should be easy to follow assuming you have made it this far without frothing at the mouth and falling into a coma. Remember that these mathematical statements are functions and as such, the 'operation' comes first followed by the variables to be operated on. If it helps make sense of the syntax, in your mind just move the operator from in front of the function to in between the variables and they will seem more familiar. This will also help keep straight the order of operation in subtraction, division and in relational functions.

```
(+  <operand1> <operand2>)
```

This is simple enough. We add the value of operand2 to operand1 and send the result to whatever function contains this function. In short, there is no '**target**' or destination implied by just this function alone. You will use it within another larger function, most likely with an "**equal**" assignment sending the result to a third variable. For example, we might do something like:

```
(= Note.Vel (+  hammer baseline))
```

in order to set the velocity of a note equal to the sum of two variables. As you might guess, one or both of these variables could also be constants or other functions such as:

```
(=  Note.Dur (+  (+  offset pastdur) (-  Event.Time lasttime)))
```

Here note duration is set equal to the sum of an offset amount and a stored duration plus the difference between the event's starting time and another stored time. Notice the way the functions stack. If you were to replace any of the **nested** functions with values, the order of the larger function's parts and mathematical operations would stay the same. Remember, CAL does not care if you use variables, constants or nested functions as arguments. It is all the same to CAL. The placement of the "**plus**" keywords and parentheses never changes. You just stack them when the functions take on nested functions.

```
(-  <operand1> <operand2>)
```

Take a close look at this function. With the addition function above, the order of the variables did not matter. Addition gives the same result adding 2+3 and 3+2 due to addition being '**commutative**'. This is not the case for subtraction. This statement reads in English '**Subtract from operand1 the value in operand2**', or simply, '**operand1 minus operand2**'. Don't get the order mixed up or you will have all kinds of problems. Otherwise, the rules for using "**-**" are the same as for using "+".

```
(*  <operand1> <operand2>)
```

This is multiplication. Here the order does not matter. Just make sure you do not overrun the range limits of your variables. The rules for using this function are as described above.

```
(/  <operand1> <operand2>)
```

Division also has a specific order of the operands. This statement reads in English '**divide operand1 by operand2**'. Keep in mind that CAL does not see fractions and does not round off. The result of division in CAL is the integer value only. You must use the "**modulo**" function below to obtain the remainder of the division. Otherwise, the rules mentioned above apply.

```
(%  <operand1> <operand2>)
```

Because CAL does not use fraction and does not round off division, we use the "**modulo**" function to obtain the fractional part of a division. The result of (% <operand1> <operand2>) is the integer number left over after subtracting '**operand2**' from '**operand1**' over and over (which is the act of division) until there is a remainder that is less than '**operand2**' and thus there can be no further subtraction without sending the result negative. This remainder is the fraction of '**operand1**' left over from the division. For example, if 23 were divided by **4**, the answer would be **5** and the remainder would be **3**. Thus, in CAL, the function (/ 23 4) would result in an answer of **5** and (% 23 4) would result in an answer of **3**. Here is how you would use this function to round of division. Note the following formula for obtaining an average.

```
(= average (/  sum count))
(if  (>=  (%  sum count) (/ count 2)) (++  average))
```

First, the average is found by dividing the sum of numbers by the total of numbers that were summed. Assume we have added 45 numbers together and the total was 22845. Therefore, '**sum**' is 22845 and '**count**' is 45. We divide the two to get the average and the answer is 507.67. Because CAL does not see fractions and will not round, the answer for the first function above will be the integer 507. However, for accuracy, we would want to round off this number to 508. The second function does that by taking the remainder of the division of 22845 by 45, which is 30, and seeing if this remainder is greater than or equal to half of count, or half of 45. If it is, then we increment '**average**' and get 508 instead of 507. If you look at a CAL program supplied with Cakewalk called '**Thin controller data.cal**', which is a program that thins out controller events by deleting every Nth event, you will see the modulo function used to locate that Nth event by counting and dividing, and whenever it sees a remainder of 0, an event is deleted. Everything said about using and setting up the other mathematical functions applies to the "**modulo**" function as well.

```
(++  <variable>)
```

Here is a special case of a combo mathematical / assignment function. This simply adds 1 to the value of '**variable**'. This is referred to as '**incrementing**' a variable and often happens in association with counting the occurrence of some kind of condition. Such things as counting the number of times a "**while**" loop has executed or tallying the number of notes encountered are handled by the "**increment**" function.

```
(--  <variable>)
```

This is the same thing as incrementing except it subtracts 1 from a variable. This is known as '**decrementing**' and has a lot of the same uses including countdown timers and so on. Keep in

mind that it is a bad idea to let a variable decrement below zero unless the variable can handle negative numbers.

```
(random  <minvalue> <maxvalue>)
```

Although this is sort of an assignment function, I place it here because it, like the rest of these mathematical functions, has no implied destination for the result and must therefore be a nested function of a larger function. What this guy does is come up with a pseudo-random number that can be any number starting with '**minvalue**' up to '**maxvalue**'. The reason it is called a **pseudo-random** number is that if you did nothing but call this function and your computer executed the exact same number of machine cycles between each number's output, the results would eventually form a repeating pattern and as such would be predictable and not truly random. In practice, however, the results are random enough for anything one might want to use the function. Mostly, it is used to generate random notes or timing offsets for making computer generated music or **humanizing** (fluctuating slightly) the starting times of notes that have been hand-entered or quantized.

2.4.8 Relational functions

Relational functions have nothing to do with the mating habits of CAL programs (disappointed? Sorry). What these functions do is test arguments for there relative values by making comparisons. I'm sure once you've seen them, they'll make sense. Using these functions does not change the values of any variables and does not result in a numerical result (well, it really does, but we don't care what that number result is). The outcome of running relational functions is either "TRUE" or "FALSE", and as far as we care, this is all we need to know about their output. Just for the record, the numerical result for a condition being "FALSE" is **0** and for being "TRUE" is any other number, usually **1**. Like the mathematical functions, these functions do not stand alone but make up the condition statements of "if" and "**while**" functions. In order for CAL to make a decision, it needs to test some condition and plan its course of action based on the outcome. We will see many examples of these statements, as conditional tests make up the heart of many CAL programs.

This first group make comparisons between two arguments and offer a result. One argument can be compared as being equal to the other, not equal to the other, larger than the other, smaller than the other or some combination.

```
(==  <operand1> <operand2>)
```

This function tests for equality between two operands e.g. a variable and a constant. Either or both operands could also be functions instead. The point is that the result will be "TRUE" if the two arguments are "**equal**" to each other and "FALSE" if they are not. Notice that the keyword ("==") is made up of two equal signs, and there is to be no space between them. If you remember from an earlier part of this document, there can be some confusion between the use of a single equal sign that means '**Store operand2 into operand1**' and these double equal signs that mean '**Test to see if operand1 is equal to operand2**'. In the first case, '**operand1**' has its value reassigned and in the second case nothing changes and only a condition is tested for. Using such a statement with an "if" function is the most common form. Here is an example:

```
(forEachEvent
   (if  (==  Event.Kind NOTE)
       (+=  Note.Vel 10)
    )
)
```

In this example we compare the value CAL has provided us in "Event.Kind" with the event kind "NOTE" to see if the current event in our "**forEachEvent**" loop is a note. If it is, then the nested function that makes up the '**then**' part of our "**if**" function is carried out. In this case, the note velocity is increased by 10. If the event had not been a note, the "**if**" condition would have failed and no action would have been taken on the event.

(!= <operand1> <operand2>)

This is the opposite of the above function. Here we test for the two operands being unequal. The result is "TRUE" if the values of the variables are different and "FALSE" if they are equal. As always, we could use a constant for one of the variables and/or nested functions for one or both of them. Here is an example of using "**!=**".

```
(forEachEvent
    (if  (==  Event.Kind CONTROL)
       (if  (!=  Control.Num 7)
         (delete)
       )
    )
)
```

Here we test each event during a "**forEachEvent**" loop to find only controller events. Having found one, we test to see if the event is not a volume controller (controller number 7) and if it isn't, we delete it. When we look at the Boolean functions, I'll show you how to make this same program fragment more efficient by combining the test for controller events and test for the controller not being number 7 into a single "**if**" statement (see in chapter 2.4.9: the logical "**AND**" function).

(< <operand1> <operand2>)

As you might guess, this statement tests for '**operand1**' being less than '**operand2**'. We might use this function to segregate items or variables as in this example:

```
(if  (<  Note.Key 64)
    (+=  Note.Vel 10)
)
```

This little fragment tests to see if the current note event has a pitch less than 64, or E3 in MIDI note form, and if so raises the note's velocity by 10. The usual rules for using constants and functions in place of variables apply.

(<= <operand1> <operand2>)

A variation on the above function, this one results "TRUE" if '**operand1**' is less than or equal to '**operand2**'. If we had to do this in long hand as part of an "**if**" function, it would look like this:

```
(if  (||  (<  variableA variableB)  (==  variableA variableB))
    (somefunction)
)
```

I have used the Boolean "**OR**" function to create the long hand version. This statement reads '**If either variableA is less than variableB or variableA equals variableB then return a result of "TRUE" and run (somefunction)**'. As you can see, the "**<=**" shorthand is superior. The same "**if**" statement above now reads:

```
(if  (<=  variableA  variableB)
  (somefunction)
)
```

which is much easier to read and work with.

```
(>  <operand1> <operand2>)
```

As the obvious complement to the above "<" function, this one tests for '**operand1**' being greater than '**operand2**'.

```
(>=  <operand1>  <operand2>)
```

We finish this group with the "**greater than or equal**" function. It works like the others shown before.

2.4.9 Boolean functions

The two Boolean functions derive from classical logic. The two operator keywords CAL uses are logical "**AND**" and logical "**OR**". In the academic world, these operators are associated with a '**truth table**' which describes the outcome of applying Boolean logic to a set of conditions. I will save you the trouble by saying it in simple English.

```
(&&  <operand1> <operand2>)
```

This is the logical "**AND**" function. As with the other relational functions, the Boolean functions are not meant to stand alone but are to be used as operands for "**if**" and "**while**" functions. Here, '**operand1**' and '**operand2**' apply to any statement that produces an outcome. Although it is possible to use a variable alone as a condition, in that its very existence defines an outcome, the most practical application is to make the '**conditions**' relational functions. First, let me explain what logical "**AND**" means. Simply put, the outcome of the "**AND**" function will be "TRUE" if and only if **both** '**operand1**' and '**operand2**' are "TRUE". Here is an example:

```
(if  (&&  (==  Note.Key basenote)  ( >  Note.Vel minlevel))
     (insert Event.Time Event.Chan NOTE (+  Note.Key 12)  Note.Vel Note.Dur)
)
```

With this example, we test to see if **both** the current note pitch is equal to a pitch number stored in '**basenote**' **and** the note has a velocity greater than the value stored in '**minlevel**'. If **both** of these conditions are met, then we insert a new note at the current note's event time and on the current note's channel with the current note's velocity and duration, but one octave higher in pitch. If either of the conditions had been "FALSE", the entire "**if**" statement would have been "FALSE" and thus, no new note would be inserted. As promised, I will now show how to make more efficient conditional statements using Boolean functions. If you recall earlier we had an example that went like this:

```
(forEachEvent
     (if  (==  Event.Kind CONTROL)
          (if  (!= Control.Num 7)
               (delete)
          )
     )
```

```
)
```

We can put both conditions in one statement using "**AND**" like so:

```
(forEachEvent
     (if   (&&   (==   Event.Kind CONTROL)  (!=   Control.Num 7))
          (delete)
      )
)
```

This makes the code a bit more compact and clear.

There is a limit as to how many individual functions you can stack in a single statement like this. At some point, CAL will give you an 'Evaluation stack overflow' error if you get too carried away.

```
(||   <operand1> <operand2>)
```

The other Boolean function is the logical "**OR**". For this statement to be "TRUE", **either** one of the operands or both must be "TRUE". The "**OR**" functions resolves to "FALSE" only if both conditions are "FALSE". You can think of "**OR**" as the inverse of "**AND**". Here is an example of an "**OR**" function:

```
(while   (||   (< count 64)   (<   time Event.Time))
     (do
          (insert time Event.Chan CONTROL 7 count)
          (++   time)
          (++   count)
     )
)
```

Ah, the "**while**" loop function! I will go into how the "**while**" loop works in chapter 2.4.10, but for now understand that a "**while**" loop executes its nested functions until the conditional statement becomes "FALSE". At that time, the loop becomes unstuck and the program continues with whatever comes after the "**while**" loop closing parenthesis. In this example, the while loop has a Boolean "**OR**" function as its conditional statement. It says that for as long as either the value of '**count**' is less than 64 "**OR**" '**time**' is less than "Event.Time", keep inserting volume events and incrementing both 'count' and 'time'. At some point, '**count**' will become greater than 64 and 'time' will exceed "Event.Time", then the "**OR**" condition will be "FALSE" and the loop will abort. This program fragment does not have much use, but serves as an example only. For the record, you can stack "**OR**" functions just as we did with the "**AND**" functions. In fact, you can mix "**AND**" and "**OR**" functions in the same stack.

2.4.10 Control flow functions

Execution of CAL programs starts at the top of the code and flows down to the bottom without any chance of being directed back up or skipping down or running sub-routines. This lack of sub-routine and jump functions is annoying at times, but not insurmountable. For the record, there are some limited ways of branching out of the '**top down**' format, but very limited. Nevertheless, these meager branching options are enough to allow the needed flexibility to generate some very powerful programs. There are two effective looping and two branching functions. The earlier described "**forEachEvent**" function can be seen as a looping function, but the real available looping function is the "**while**" function. Furthermore, CAL has the branches "**if**" and "**switch**".

However, we will start with the nesting control function "**do**" and the external program spawning function "**include**", which is the closest thing, we have to a user-friendly '**gosub**' function. Finely we will touch on the "**DLL**" function, which is an "**include**"-type function that operates on the "**DLL**" programs that exist to support Windows and Windows applications.

```
(do  <expression1> [[<expression2>]…])
```

The "**do**" function is a nest builder. It combines multiple functions and nests them so they will fit into places where CAL expects to see only one function. This is why we start CAL programs with a "**do**". A CAL program is simply one big function! The same is true for the functions that go within "**if**" functions, "**while**" and "**forEachEvent**" loops, indeed anywhere that has requirements for a single function but more needs to get done than will fit in one function. Here is a specific example:

```
(forEachEvent
    (do
        (somefunction)
        (somefunction)
        (somefunction)
        (anotherfunction)
        (soforth)
        (andsoforth)
    ) ; end of do
) ; end of forEachEvent
```

See how all of the functions, which come after the "**do**", are grouped into one nest because the "**forEachEvent**" function can have but one function as an action. That one function is "**do**". Any of the functions nested within the "**do**" can likewise be "**do**" functions with their own nested functions. Here is a place where you might think a "**do**" is needed but isn't:

```
(forEachEvent
    (if  (==  Event.Kind NOTE)
        (while  (<  Note.Key 64)
            (if  (>  Note.Vel 40)
                (+  Note.Key 10)
                (+  Note.Key 20)
            ) ; end of if
        ) ; end of while
        (delete)
    ) ; end of if
) ; end of forEachEvent
```

This is another useless code fragment, but proves a point. We do not need any "**do**" functions because the "**forEachEvent**" sees only one function, the "**if**". The "**if**" sees only one function, the "**while**" loop for its '**then**' part. The "**while**" loop sees only one function: the last "**if**". This "**if**" sees only one function for its '**then**' part and only one function for its '**else**' part. Afterwards, the first "**if**" has only one function, the "**delete**" for its '**else**' part. This is an extreme example, but illustrates that you must think about your "**do**" placements as each nested function takes up room in CAL and leaves less for calculations. If the nesting levels become too encumbered, it has been my experience that you can elicit an 'Evaluation stack overflow' error at some point from CAL. There is one qualifier to the above statements. I have had times when building multiple "**if**" functions, that

on some rare occasions one of the nested "**if**" statements will not execute. I solved the problem just by adding an otherwise useless "**do**" and nested the problem "**if**" statement under it. I really did not need it because there was technically only one function there. It seems that the "**do**" somehow sharpened CAL's view of the compound statement. Otherwise I cannot explain why the "**do**" was necessary, it just worked! For a more detailed discussion of this problem, see the section chapter 5.6.2: 'Hidden traps in the CAL editor'.

Loops are chunks of code that keep being executed over and over until some condition tells them to stop. We have already shown the "**while**" function, but now is the time to get knee-deep into them.

```
(while  <condition> <action>)
```

The "**while**" loop is very useful. It can used to execute a nested function for as long as '**condition**' is "TRUE". You can run "**while**" loops within a "**forEachEvent**" loop, you can run "**forEachEvent**" loops within a "**while**" loop, you can even run "**while**" loops within other "**while**" loops. The '**condition**' can be any valid function or group of nested functions so long as it will resolve to a "TRUE" or "FALSE" result. There are so many uses for the "**while**" loop that I hardly know where to begin giving examples! We looked at one example when we demonstrated the "**OR**" function. Here is another:

```
(while (<  looptime stoptime)
    (do
        (= newa (+ ref (/  (*  percent change) 100)))
        (insert (+  notetime looptime) chan CONTROL 7 newa))
        (+=  looptime gap)
        (=  percent (/ (*  looptime 100) stoptime))
    ) ; end of do
) ; end of while
```

This is a simplified fragment from a program Glen wrote that performs automatic cross fades for the duration of a note from the first track to the second track. For each note, it starts again in track one. He has removed many of the contents of the loop and just left the fading in one track to explain the "**while**" function.

This "**while**" loop is within a "**forEachEvent**" loop so that it runs for each note in the selected part of the sequence. First, we have the condition for the loop. So long as '**looptime**' is less than '**stoptime**', the loop will run.

We use "**do**" to create a nest for the 4 functions that follow so that "**while**" will see only one function in its nest just like "**forEachEvent**" as discussed above. The first nested function calculates a value for '**newa**' by multiplying '**percent**' and '**change**', dividing the result by 100 then adding it to the offset '**ref**'. This value will be the value of our volume controller event, which we insert in the next line at time '**notetime**', plus '**looptime**'. We then add '**gap**' to '**looptime**' to point to the place we want to insert the next volume event, and then calculate a new value for percent, which is derived by seeing how far along we, are in '**looptime**' relative to '**stoptime**'.

Now we go back and test the condition again to see if we can run another "**while**" loop or if the conditions have been met and we can exit the loop. When '**looptime**' fails to be smaller than '**stoptime**', the loop ends. It is important to keep in mind that if the '**condition**' is false the first time CAL looks at it, the "**while**" loop will never run at all. This is useful because there may be times when you will want to run the loop only if '**condition**' exists and bypass the loop if '**condition**' doesn't exist. By the way, you can use "**while**" loops to build on-line help systems for your CAL programs. Check out in chapter 4.6: 'Implementing On-Line help in a CAL program'.

'**Branching**' functions allow us to skip over some of the code that we do not need on that occasion and go directly to some code that we do need. The rules for using the two branching functions are to be strictly followed or your program can do some strange stuff!

```
(if  <condition> <true-expression> [<false-expression>])
```

In many examples I have mentioned the "**if**" function. I will go now into more detail of this function. It is the well known '**if then else**' function which exists in all programming languages:

```
(if  (condition)  (thenfunction)  (elsefunction))
```

This form is just fine if the '**thenfunction**' and, if used, the optional '**elsefunction**' are simple and not made up of nested functions. Otherwise, the more readable form would be

```
(if  (condition)
     (thenfunction)
     (elsefunction)
)
```

Needless to say, the '**thenfunction**' and '**elsefunction**' functions can be compound nested functions as well. Look at this monster "**if**" function:

```
   (if  (&& (== Event.Kind NOTE) (> Note.Key lownote))    ; first if
     (do                                                  ; first do
(if  (!= Note.Key lastnote)                          ; second if              (do
                                                ; second do
           (if  (< Note.Dur length) (= Note.Dur length)) ; third if
                                                ;   on one line
           (++ count)
         ) ; closing of second do
         (if  (>= count limit) (+ lownote offset))       ; forth  if on
                                                ;  one line
       ) ; end of second if
       (insert Event.Time Event.Chan NOTE lownote Note.Vel length)
     ) ; end of first do
     (= count 0)
  ) ; end of first if
```

Again this code fragment does not really do anything useful, it is just a bunch of stuff in order to demonstrate the form of a compound "**if**" statement. The first "**if**" statement has a condition made of the Boolean function "**AND**" and two relational functions, both of which must be "TRUE" for the "**if**" function to execute. This "**if**" has a '**then**' part that is defined by all statements within the first "**do**" function. This "**do**" function combines two functions, the second "**if**" and the "**insert**" function. As it happens, the second "**if**" uses a "**not equal**" as its conditional, It nests a '**then**' part in the form of the second "**do**" and an '**else**' part, the forth "**if**" function. The second "**do**" nests the third "**if**" function and an "**increment**" function. The third and forth "**if**" functions are on one line in that they have only a short conditional statement, a single '**then**' function and no '**else**' part.

Do you notice how the functions at the same indention point within an "**if**" function make up the '**then**' and '**else**' parts? Notice also that the functions at the same indention point within a "**do**" function are all separate functions that share the same nest. Can you see how the closing parenthesis of a compound function is always at the same indention point as the corresponding opening parenthesis that resides a few lines above? These are important conventions that make good programming habits and save hours of frustration if things need to be changed later on. Remember,

the '**then**' part is mandatory, and the '**else**' part is optional. You cannot technically have an "**if**" function with an '**else**' part and no '**then**' part. This is of little consequence, because an "**if**" statement condition can usually be rewritten to make it unnecessary.

If for some reason you have built a condition so complex or wish to keep it readable from some particular point of view, and the action of the "**if**" function as written need only act upon events other than those filtered by the condition (the condition selects events that must be bypassed and everything else needs to have something done to it), then you can make a do-nothing '**then**' part using the relatively undocumented "NIL" keyword and have the '**else**' part do all of the work. The form might look something like this:

```
(if  (&& (==  Event.Kind NOTE)  (||  (>  Note.Vel 40)  (>=  Note.Dur 10)))
    NIL
    (delete)
)
```

In this example, we set up a rather complicated condition that an event must be a note event and have either a velocity above 40 or a duration as long or longer than 10 ticks. If these conditions are true, we do nothing. For all other events and for notes that do not meet these conditions, we delete them. Notice that "NIL" is not in parentheses. I don't know why this is so, it just works that way. You never need to put "NIL" in parentheses, I guess because as far as CAL is concerned, it does not exist! Technically, we could rewrite the above condition so that the '**then**' part could hold the "**delete**" function like this:

```
(if (|| (!= Event.Kind NOTE) (&& (<= Note.Vel 40) (< Note.Dur 10))) (delete))
```

This condition says that if an event is either not a note or is a note that has both a velocity less than or equal to 40 and a duration less than 10, it will be deleted. Both statements do the same job, but the second one is simply better form. However, if for continuity reasons it behooves you to create the "**if**" condition in the first form, the "NIL" keyword is available to you.

```
(switch  <index> <case1> <case1result> [<case2>
<case2result> …])
```

The "**switch**" function is a very useful. Therefore it is important that you understand how and when to use it. This is the full form as you would use it in a program:

```
(switch value
       case_1     (function_1)
       case_2    (function_2)
      …
       last_case    (last_function)
)
```

What "**switch**" does is allow you to select a chunk of code to run based on the contents of '**value**'. The '**case**' parts are simply the various possible values that '**value**' can have, and the '**function**' part is what ever you wish to happen if '**value**' equals '**case**'. CAL evaluates the '**value**' part, which may be a variable or a function or nested functions or anything you want to use. Whatever the outcome is, even if it's a "TRUE" or "FALSE" outcome, CAL will scan the list of '**case**' values until it finds a match and then run the associated '**function**' and only that function. Having done this, CAL exits the "**switch**" function (often referred to as a '**Switch Tree**') and continues with whatever follows the "**switch**" closing parenthesis.

Of course, using a '**Switch Tree**' for a "TRUE"/"FALSE" '**value**' wouldn't make much sense because there are only two possible outcomes, and an "**if**" function is all you need to select a course of action based on two outcomes. Indeed, it makes no sense to use "**switch**" at all unless you wish

to select from three or more possible outcomes of '**value**'. As you might suspect, '**function**' can be a simple function or a compound function of any size, just as long CAL sees only one function in each '**case**'. All you need is a "**do**" to nest together everything you want to have run and you can place entire programs in each '**case**'. In fact, that's one of the best purposes of a '**Switch Tree**'. See the chapter 4.5: 'Switch Tree Menus', for an example of a menu switch function.

```
(include  <filename>)
```

The "**include**" function causes the CAL program named in double quotes to be loaded and run at that point in the program containing the "**include**". After this spawned program ends, execution continues with the code that follows the "**include**" statement. It is as if the code in the '**included**' program had been pasted into the calling program in place of the "**include**" statement like a "**GoSub**" routine.. A good example is:

```
(do
    (include "NEED20.CAL")
    (declare your variables)
    (and continue as usual)
)
```

Examples of include programs can be found in the MIDI-Kit library.

You can use "**include**" to add in to all of certain types of programs some feature they all should have in common such as correcting note overlaps or removing slurred notes or whatever. See chapter 4.7 'Building an include library' for insight into using included programs.

```
(DLL <"filename"> <procedurename> <argument1> .. <argument
N>)
```

The best thing I can tell you about this function is use it as less as possible, unless you are an experienced Windows programmer. You need the documentation for using 32-bit Windows "**DLL**" (Dynamic Link Libraries) programs.

Briefly, this function calls the "**DLL**", file name listed in double quotes (without the extension "**DLL**") and requests execution of the service '**procedure name**' within that "**DLL**" file. Both of these names must be in quotes. After the 'procedure name', you supply whatever arguments the service requires in order to carry out its function. The (DLL) function will only accept two[15] arguments. If you leave out an argument you are likely to corrupt the system stack and thus likely crash your session of Windows, or at the very least make CAL and/or Cakewalk go south. If you supply more than two, those extra arguments will pass garbage instead of the data in those extra arguments to the "**DLL**". Here is a simple example:

The program uses a Windows system DLL to sound a beep..

```
(do
; sound a beep
(string sysdll "c:\windows\system\User32.dll")
(DLL sysdll "MessageBeep")
) ; end of do
```

[15] This is not according to the CAKEWALK Pro Audio help files. Here is stated that more than 2 arguments can be supplied. You **must** supply the required number of arguments for the specific procedure of the DLL program.

If you really want to use this function, see also chapter 5.10: 'Helpful Tips for Using the DLL Function'.

2.4.11 Musical time functions

There are two ways of keeping time in a sequence recognized by Cakewalk and CAL. The first is in terms of **'measure**', '**beat**' and '**tick**' called '**M:B:T**' time and the other is in the form of a 32-bit number referred to as '**raw**' time, stored in a "dword" variable.

To put it simply, raw time is a count of the number of clock ticks from the start of a sequence to the end.

'**M:B:T**' time is that same number divided by the number ticks per beat and that amount further divided by the number of beats in the current song's meter. The result is formatted in standard musical time notation with the remainder number of ticks included for accuracy.

At '**raw**' time zero, the musical time is 1:1:0 or '**measure**' = **1**, '**beat**' = **1**, '**tick**' = **0**. Note that you cannot have a measure or beat number 0. This is convention as established by hundreds of years of musical tradition and Cakewalk has no intention of flying in the face of it.

What Cakewalk adds is the '**tick**' component. Because events must be precisely located within a musical score, the ticks provide the necessary resolution without having to resort to stuff like 256th notes. Can you imagine how obnoxious such notes would be on paper with all of those little flags on the '**Staff**'?

By using ticks and setting the number of ticks per beat or quarter note, a sequence can record human feel without having to note it in the score, just like the real musicians play.

The constant "TIMEBASE" makes the user's setting of the number of ticks per beat available to CAL for calculations and evaluations. The default value is 120 ticks per beat but this can be changed from the front panel. The user can also set the meter from the front panel and thus select $^4/_4$ or ¾ or $^7/_8$ or whatever meter they want. These settings are then used to calculate the '**M:B:T**' time from '**raw**' time and vice versa. To facilitate this conversion from human to computer time keeping, CAL has provided us with a set of conversion functions. If you recall our discussion of the "**getTime**" function, which allows the user to input a time in '**M:B:T**' format and let CAL store it as a double word in '**raw**' time format (at least in versions other than 7 and 8), we saw an example of a time keeping conversion function. Here are the rest.

```
(meas <rawtime>)

(beat <rawtime>)

(tick <rawtime>)
```

These functions take the '**raw**' time, or a function that resolves to a value of '**raw**' time and calculates the proper number of measures, beats or ticks as they would appear in the "Now" window on the front panel. If you wanted to display this information to the user in a dialog box at some point in the program, you could do something like this:

```
(pause "The time is " (meas Event.Time) ":" (beat Event.Time) ":" (tick
Event.Time) " at event " (index))
```

Assuming that the "Event.Time" is, say, 22368, the "TIMEBASE" and meter settings are 120 and $^4/_4$ and the current event being scanned by the "**forEachEvent**" loop is number 2245, the "**pause**" dialog box would display:

```
The time is 47 :3 :48 at event 2245
```

(makeTime <measure> <beat> <tick>)

This is the reciprocal to the above conversion functions. Like "**getTime**", this function takes values (or functions resulting in values) for '**measure**', '**beat**' and '**tick**' and returns a raw time. Like math functions, these functions are meant to be part of a larger function where the resulting calculations can be output or assigned to some variable. Check this out:

```
(insert (makeTime measure beat tick) channel WHEEL value)
```

Here we have inserted a pitch wheel event at a time calculated from separate measure, beat and tick variables.

2.4.12 Miscellaneous functions

(error)

This function can be used to terminate your program if an error occurs. It generates a dialog box, stating that the 'Program called (error)', and terminates CAL.

(exit)

If you look at the code that makes up the '**need20.cal**' program that installs with Cakewalk, you will see an undocumented function called "**exit**". You don't see it anywhere else or see mention of it anywhere. You would think the developers of CAL were ashamed of it or something. It can be very useful as demonstrated in '**need20.cal**'. When CAL hits this keyword, the program aborts and all CAL activity ends. This can be very useful if you want to auto-exit after testing true for some fatal condition such as there being no events in range that qualify for whatever you were going to do to them.

The result of calling "(exit)" is the presentation of a dialog box, stating that the 'CAL Error 021 Program called (exit)' and the termination of CAL.

(delay <time>)

This function doesn't do anything except suspend execution of the CAL program by the number of milliseconds (1000ths of a second) in the value '**time**'. This is a way of making CAL stop what it's doing long enough for a human to read some message or react to a condition or perhaps it can be used as some kind of time keeping function causing some function to run every so often. Personally, I've never had many occasions to use it, but you might. As an example, let's say we wanted to scan a part of a sequence and give the user time to make notes of what's going on like this:

```
(forEachEvent
     (if (== Event.Kind NOTE)
         (do
             (message "Current Note Pitch Is " Note.Key)
             (delay 2000)
         )
     )
)
```

In this example we call a "**forEachEvent**" loop and use "**if**" to make sure we only look at "NOTE" events. We display on the status bar at the bottom of the screen the pitch number of the note we

have found, delay for 2 seconds to give the user time to write this number down, then go on to the next event in the loop.

```
NIL
```

This undocumented feature is not a function exactly, and it never needs to be in parentheses. It is just a placeholder. If you have a CAL function that requires an argument in some part of the expression and you don't want there to be one, you can use "NIL" as a substitute. That usage was illustrated earlier in the overview of the "**if**" function. This is about the only place I can think of to use it, that CAL versions 2 and above don't require the "NIL" as part of the code the way earlier versions of CAL did.

2.4.13 Menu functions

The following CAL functions emulate the '**Menu Bar**' commands that are available to the user from the drop-down lists at the top of the Cakewalk screen.

There are several versions for most of the '**Edit**' functions. First, there are the ones from Cakewalk version 3 which I will call CAL30 functions and those from Cakewalk version 4 and above, which I will call CAL40 functions[16]. The difference is the argument list in the CAL40 functions reflecting the **assumption** that the "From" and "Thru" markers are set elsewhere and so no provision is made in the argument list for them, nor for the '**Use Event Filter**' yes/no check box argument. Along with this change there are a few CAL40 functions that have expanded argument lists that reflect the additional options available in versions 5 and 6 such as the audio and clip editing features. Finely, there are some '**Track**' menu functions that are new as of version 7. Version 8 doesn't appear to have changed CAL from the version 7 implementation.

You will likely notice that, for the most part, if you know how to use the Cakewalk drop-down menu commands, you can use the CAL functions also. There are noticeable exceptions. One is the use of the editing and interpolating filters. These can be a bit of a pain in the butt. However, in earlier versions of CAKEWALK you could use the '**Recording Cal Macros**' to generate the prototype code for setting up these filters which then require only a bit of editing to make them usable in a program. In SONAR this is not available anymore. Therefore, I give more information about these functions. The 40-function take advantage of the filter settings of SONAR, therefore, I recommend to use them instead of the 30-functions.

Another is the bug that exists in Cakewalk version 6 which prevents the "**EditInterpolate**" function from acknowledging the "From" and "Thru" markers. Instead, the function will operate on the entire track at once. As a side effect of this bug, after the function runs, the values stored in the markers are dumped and replaced with the times corresponding to the start and end points of the sequence respectively. There is a work-around for this problem. For details see chapter 5.4 'Forcing Version 6 (EditInterpolate) To Use Markers'.

One of the most important aspects of using these MENU functions is the limitation that they will **not** run within a "**forEachEvent**" loop. This makes sense if you think about it, but I mention it here so there is no misunderstanding as to the nature of these functions. This is the only restriction. They work very well and are quite useful in "**while**" loops and branching functions like "**if**".

[16] The new versions have a '**40**' suffix in their name, for example, "**EditCut40**" is the new version of the "**EditCut**" function.

2.4.13.1 EDIT Menu Functions

(EditControlFill [<from> <thru> <ctrl> <chan> <beg> <end>])

Fills the selected range '**from'** till '**thru**' with controllers '**ctrl**' in channel '**chan**' starting with begin value '**beg**' and ending with value '**end**'.

'**chan**' is from 1..16.

Begin value '**beg**' of the controllers, and end value '**end**' must be between 0 and 127.

If you omit all parameters, the dialog box will pop up.

(EditCopy [<from> <thru> <events> <filt> <tempos> <meters> <markers>])

This is the CAL 3.0 compatible function corresponding to the '**Copy**' command. It copies the selected range '**from**' till '**thru**' to the clipboard.

You can select what will be copied by setting the parameters: '**events**', '**filt**' (use event filter), '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

If the 'use event filter' parameter '**filt**' is "TRUE", then the '**event filter**' setting will be applied.

If you omit all parameters, the dialog box will pop up.

(EditCopy40 [<events> <tempos> <meters> <markers> <audio> <clips>])

This is the CAL 4.0 function corresponding to the '**Copy**' command. It copies the selection to the clipboard.

You can select what will be copied by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

'**audio**' "TRUE" means: '**Split Audio Events**';

'**clips**' "TRUE" means: '**Copy Entire Clip As Linked Clip**'.

The settings of the range ("From" and "Thru"), and the '**event filter**' setting will be applied.

In versions of CAL between 4.0 and 6.0 some parameters are missing. The syntax is:

```
(EditCopy40 [<events> <tempos> <meters> <markers>])
```

If you omit all parameters, the dialog box will pop up.

(EditCut [<from> <thru> <events> <filt> <tempos> <meters> <markers> <hole>])

This is the CAL 3.0 compatible function corresponding to the '**Cut**' command. It cuts the selected range '**from**' till '**thru**', and copies it to the clipboard.

You can select what will be copied by setting the parameters: '**events**', '**filt**' (use event filter), '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

If the 'delete holes' parameter '**hole**' is "TRUE", then the created holes will be deleted.

If the 'use event filter' parameter '**filt**' is "TRUE", then the '**event filter**' setting will be applied.

If you omit all parameters, the dialog box will pop up.

(EditCut40 [<events> <tempos> <meters> <markers> <hole> <split> <align>])

This is the CAL 4.0 function corresponding to the '**Cut**' command. It cuts the selection and copies it to the clipboard.

You can select what will be copied by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

'**hole**' "TRUE" means: '**Delete Holes**';

'**split**' "TRUE" means: '**Split Audio Events**';

'**align**' "TRUE" means: '**Align To Measures**'.

The settings of the range ("From" and "Thru") and the '**event filter**' setting will be applied.

Between CAL 3.0 and CAL 6.0 there are less parameters:

```
(EditCut40 [<events> <tempos> <meters> <markers> <delhole>])
```

If you omit all parameters, the dialog box will pop up.

(EditDelete40 [<events> <tempos> <meters> <markers> <hole>[<split> <align>]])

This is an undocumented CAL 4.0 function. You can select what will be deleted by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE". If 'hole' is FALSE, omit 'split' and 'align'.

'**hole**' "TRUE" means: '**Delete Holes**';

'**split**' "TRUE" means: '**Split Audio Events**';

'**align**' "TRUE" means: '**Align To Measures**'.

If hole is "FALSE", then omit '**split**' and '**align**'.

If you omit the parameters the dialog box will pop up.

(EditFitImprov [<referencetrack>]

This is the CAL 3.0 compatible function corresponding to the '**Fit Improvisation**' command. It changes tempos according to the reference track. The reference track is a single clip with a single note on each beat boundary.

'**referencetrack**' the track number (0 …255).

If you omit the parameter, the dialog box will pop up.

(EditFitImprov40)

This is the CAL 4.0 version corresponding to the '**Fit Improvisation**' command.

The CAL 4.0 version does not have a parameter. It assumes that the reference track is selected.

(EditFitToTime[17] [<from> <thru> <newthru> <method> <stretch>])

This is the CAL 3.0 compatible function corresponding to the '**Fit to Time**' command. It fits the range into the specified time. It inserts tempo changes or modifies event times in your MIDI sequence.

'**from**' the from value;

'**thru**' the thru value;

[17] In SONAR this function has a bug. It returns the message '**invalid value**'.

'**newthru**' the new thru value;

'**method**' event times or tempo map, "TRUE" means: '**Event Times**';

'**stretch**' "TRUE" means: '**Stretch Audio**'[18].

Below CAL 4.5 there is one parameter less. The syntax is:

```
(EditFitToTime[19] [<from> <thru> <newthru> <method>])
```

If you omit all parameters, the dialog box will pop up.

(EditFitToTime40[20] [<newthru> <method> <stretch>])

This is the 4.0 function corresponding to the '**Fit to Time**' command. It fits the selected range into the specified time. It inserts tempo changes or modifies event times in your MIDI sequence.

'**newthru**' the new thru value;

'**method**' event times or tempo map, "TRUE" means: event times;

'**stretch**' "TRUE" means: '**Stretch Audio**'.

For CAL 4.0 there is one parameter less. The syntax is:

```
(EditFitToTime40[21] [<newthru> <method>])
```

If you omit all parameters, the dialog box will pop up.

(EditGrooveQuantize [<from> <thru> <filt> <res> <window> <time> <dur> <vel> <owm> <groove>])

This is the CAL 3.0 compatible function corresponding to the '**Groove Quantize**' command. It applies the Groove Quantize command to the range '**from**' till '**thru**'. Settings can be:

'**filt**'	if "TRUE" the event filter will be used;
'**res**'	resolution in ticks;
'**window**'	sensitivity in percentage;
'**time**'	percentage of time (0..100);
'**dur**'	percentage of duration (0…100);
'**vel**'	percentage of velocity (0…100);
'**owm**'	'**Out Of Window Mode**':
	0 = do not change;
	1 = quantize to resolution;
	2 = move to nearest;
	3 = scale time;

[18] The '**Stretch Audio**' is limited to values between 25 % and 400 %.
[19] In SONAR this function has a bug. It returns the message '**invalid value**'.
[20] This is an undocumented function. In SONAR it has a bug. It returns the message '**invalid value**'
[21] This is an undocumented function. In SONAR it has a bug. It returns the message '**invalid value**'

'**Groove**' '**Groove Source**' name between quotes[22].

If you omit all parameters, the dialog box will pop up.

(EditGrooveQuantize40 [<res> <window> <time> <dur> <vel> <owm> <file> <pattern> <NLAonly>]

This is the CAL 4.0 function corresponding to the '**Groove Quantize**' command. Settings can be:

'**res**' in note values (0 is whole note, 1 is halve note, etc.)[23];

'**window**' sensitivity in percentage;

'**time**' percentage of time (0..100);

'**dur**' percentage of duration (0…100);

'**vel**' percentage of velocity (0…100);

'**owm**' '**Out Of Window Mode**':

0 = do not change;

1 = quantize to resolution;

2 = move to nearest;

3 = scale time;

'**file**' '**Groove File**' name between quotes;

'**pattern**' '**Groove Pattern'** name between quotes;

'**NLAonly**' "TRUE" means: '**Notes, Lyrics, Audio Only**'.

If both '**file**' and '**pattern**' are empty – indicated by: "" – the clipboard will be used as source.

If you omit all parameters, the dialog box will pop up.

(EditInterpolate [<nodialog>])

This function presumes, that the search (1) and replace (2) filters have already been set as desired.

'**nodialog**' **any value** means no dialog box, and the filter settings are used;

omitted means the dialog box are presented, and the filter settings by CAL functions are ignored[24].

See also chapter 5.4: Forcing Version 6 (EditInterpolate) To Use Markers.

(EditLength [<from> <thru> <filt> <percent> <start> <dur>])

This is the CAL 3.0 compatible function corresponding to the '**Length**' command. It stretches or shrinks the range ('**from**' till '**thru**') according to the parameter settings.

---

[22] "<clipboard>" means that the clipboard is used as source.

[23] To be checked.

[24] For CAL below version 7.0 the dialog box presents the filter settings as set with the appropriate CAL functions.

'**percent**' percentage to stretch or shrink;

'**filt**' "TRUE" means: the event filter will be used;

'**start**' "TRUE" means: start time will be modified;

'**dur**' "TRUE" means: duration will be modified.

If you omit all parameters, the dialog box will pop up.

(EditLength40 [<percent> <start> <dur> <stretch>])

This is the CAL 4.0 function corresponding to the Length command. It stretches or shrinks the according to the parameter settings.

'**percent**' percentage to stretch or shrink;

'**start**' "TRUE" means: start time will be modified;

'**dur**' "TRUE" means: duration will be modified;

'**stretch**' "TRUE" means: '**Stretch Audio**'[25].

Between CAL 4.0 and CAL version 6.0 the '**Stretch Audio**' parameter is not valid.

The settings of the range ("From" and "Thru") and the '**Event Filter**' settings will be applied.

If you omit all parameters, the dialog box will pop up.

EditPaste [<to> <rep> <mode> <events> <tempos> <meters> <markers>])

This is the CAL 3.0 compatible function corresponding to the '**Paste**' command. It pastes the clipboard in a track at the time **to**.

You can select what will be pasted by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

'**rep**' is the number of repetitions;

'**mode**'[26] 0 means: blends new material with current material;

1 means: replaces existing material with the pasted material;

2 means: slides old material over and makes room for the new pasted material.

If you omit all parameters the dialog box will pop up.

(EditPaste40 [<align> <link> <to> <totrack> <onetrack> <rep> <gap> <newclip> <events> <tempos> <meters> <markers> <mode> <split>])

This is the CAL 4.0 function corresponding to the '**Paste**' command. It pastes multiple tracks, cut or copied to the clipboard, in the same order starting in destination **totrack** at the time **to**.

You can select what will be pasted by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

'**align**' "TRUE" means: '**Align To Measures**';

'**link**' "TRUE" means: '**Link Repetitions**';

'**onetrack**' "TRUE" means: '**Paste To One Track**';

'**rep**' is the number of repetitions;

'**gap**' this is the gap in '**rawtime**' between the pasted repetitions;

[25] The '**Stretch Audio**' is limited to values between 25 % and 400 %.

[26] This parameter is only used for CAL 3.0. All other versions of CAL ignore the value.

'**newclip**' "TRUE" means: '**Paste As New Clip**';

'**mode**'[27] 0 means: blends new material with current material;

1 means: replaces existing material with the pasted material;

2 means: slides old material over and makes room for the new pasted material.

'**split**' "TRUE" means: '**Split Audio Events**'.

Between CAL 3.0 and 6.0 there are less parameters. The syntax is:

```
(EditPaste40 [<to> <totrack> <onetrack> <rep> <gap> <newclip> <events> <tempos>
<meters> <markers>])
```

If you omit all parameters, the dialog box will pop up.

(EditPastToTrack [<to> <rep> <mode> <events> <tempos> <meters> <markers> <track)])

This is the CAL 3.0 compatible function corresponding to the '**Paste**' command. It pastes the clipboard in a track at the time **to**.

You can select what will be pasted by setting the parameters: '**events**', '**tempos**', '**meters**', and '**markers**' on "TRUE" of "FALSE".

'**rep**' is the number of repetitions;

'**mode**'[28] 0 means: blends new material with current material;

1 means: replaces existing material with the pasted material;

2 means: slides old material over and makes room for the new pasted material.

'track' the track number (0..15).

If you omit all parameters the dialog box will pop up.

(EditQuantize [<from> <thru> <filt> <percent> <times> <dur>])

This is the CAL 3.0 compatible function corresponding to the '**Quantize**' command. It quantize the range ('**from**' to '**thru**').

'**filt**' "TRUE" means: the event filter will be used;

'**percent**' the percentage that indicates the placement of **out-of-time** events in perfect timing;

'**times**' "TRUE" means: '**Event Times'** will be effected;

'**dur**' "TRUE" means: duration's will be trimmed on the resolution marks.

If you omit all parameters, the dialog box will pop up.

(EditQuantize40 [<res> <percent> <times> <dur> <swing> <window> <offset> <NLAonly> <stretch>])

This is the CAL 4.0 function corresponding to the '**Quantize**' command.

'**res**' resolution in ticks;

'**percent**' the percentage that indicates the placement of **out-of-time** events in perfect timing;

'**times**' "TRUE" means: '**Event Times'** will be effected;

'**dur**' "TRUE" means: duration's will be trimmed on the resolution marks;

'**swing**' swing in percentage;

---

[27] This parameter is only used for CAL 3.0. All other versions of CAL ignore it.
[28] This parameter is only used for CAL 3.0. All other versions of CAL ignore the value.

'**window**' percentage of the window, where will be looked at out of place events;

'**offset**' off set in ticks of the resolution marks;

'**NLAonly**' "TRUE" means: '**Notes, Lyrics, Audio Only**';

'**stretch**' "TRUE" means: '**Stretch Audio**'.

Between version 4.0 and 6.0 there is one parameter less. The syntax is:

```
(EditQuantize40 [<res> <percent> <times> <dur> <swing> <window> <offset>
<NLAonly>])
```

If you omit all parameters, the dialog box will pop up.

(EditRetrograde [<from> <thru> <filt>])

This function reverses the order of events in the selected part (determined by " From" and "Thru") of a track.

'**filt**'"TRUE" means: the event filter will be used.

If you omit all parameters, the dialog box will pop up.

(EditRetrograde40)

This is the CAL 4.0 compatible function corresponding to the '**Retrograde**' command. It has no arguments, but uses the values "From", "Thru", and the filter settings.

(EditSlide [<from> <thru> <filt> <amount> <units>])

This function slides (shifts) the selected events (between 'from', and 'thru').

'**filt**' "TRUE" means: the event filter will be used;

'**amount**' the number of units;

'**units**' the type of units: **Still to be filled in**

If you omit all parameters, the dialog box will pop up.

(EditSlide40 [<amount> <ticks> <events> <markers>])[29]

This is the CAL 4.0 function corresponding to the '**Slide**' command. It slides (shifts) the by the user selected range. You can select what will be shifted by setting the parameters: '**events**', '**tempos**', and '**markers**' on "TRUE" of "FALSE".

'**amount**' the number of units;

'**ticks**' is "TRUE", then '**amount**' is treated as ticks; otherwise it is treated as measures.

If you omit all parameters, the dialog box will pop up.

(EditTranspose [<from> <thru> <filt> <amount> <diatonic>])

This function transposes the selected notes (between 'from', and 'thru').

'**filt**' "TRUE" means: the event filter will be used;

'**amount**' the number of units;

'**diatonic**' "TRUE" means: diatonic transposition.

If you omit all parameters, the dialog box will pop up.

[29] The parameters '**events**' and '**markers**' are added from CAL version 6.0.

(EditTranspose40 [<amount> <diatonic> <audio>])[30]

This is the new CAL 4.0 function corresponding to the '**Transpose**' command.

'**amount**' the number of units;

'**diatonic**' "TRUE" means: diatonic transposition;

'**audio**' "TRUE" means: audio will be transposed.

If you omit all parameters, the dialog box will pop up.

(EditVelocityScale [<from> <thru> <filt> <beg> <end> <unitspct>])

This function lets you create crescendos (volume swells) and decrescendos on those instruments that respond to MIDI velocity. The velocity is changed in the range '**from**' till '**thru**' with starting value '**beg**' and end value '**end**'.

'**filt**' "TRUE" means: the event filter will be used;

'**unitspct**' "TRUE" means that '**beg**' and '**end**' are percentages.

If you omit all parameters, the dialog box will pop up.

(EditVelocityScale40 [<beg> <end> <unitspct])

This is the CAL 4.0 version of '**VelocityScale**' command. It scales the velocity of the notes within the selected range.

'**unitspct**' if "TRUE" indicates that '**beg**' and '**end**' are percentages, otherwise if "FALSE" that they are velocity values.

If you omit all parameters, the dialog box will pop up.

(ResetFilter <type> <everything>)

Whenever you use one of the EDIT menu functions, you need to let CAL know what events to include in the action. The CAL30 functions (except "EditInterpolate") give you the option of placing a "FALSE" value in the '**filt**' parameter thus ignoring the filter setting. The edit will affect all events in the range. The CAL40 functions do not have this argument and so should have at least this '**ResetFilter**' function run first to give it predictable access to events.

'**type**' indicates the kind of filter being set.
0 means: '**Use Event Filter**';
1 means: the Interpolate Event search filter;
2 means the Interpolate replace filter.

'**everything**' "TRUE" corresponds to '**Edit Select All**'.

(SetFilterKind <type> <kind> <include>)

The next filter setup function selects the actual event kind to be effected. If the above '**ResetFilter**' function's '**everything**' parameter "FALSE", then this function would select the events to be included. If '**everything**' was "TRUE", then this function would be used to tell CAL which events to exclude. If several event kinds are to be included or excluded by this method, a '**SetFilterKind**' function must be run for each event kind.

'**type**' indicates the kind of filter being set.
0 means: '**Use Event Filter**';
1 means: the Interpolate Event search filter;
2 means the Interpolate replace filter.

[30] The parameter '**audio**' is added from CAL version 6.0.

'**kind**' is one of the following kind-of-event constants:

"CHANAFT";
"CONTROL";
"KEYAFT";
"LYRIC" (new from 3.0);
"MCI" (new from 3.0);
"NOTE";
"PATCH";
"SYSX" (new from 3.0);
"TEXT" (new from 3.0);
"WAVE" (new from 3.0);
"WHEEL".

'**include**' "TRUE" means to include this kind of event.

(SetFilterRange <type> <range> <inrange> <min> <max>)

This function is a bit odd for a number of reasons I will try to explain in a moment.

'**type**' indicates the kind of filter being set.
0 means: '**Use Event Filter**';
1 means: the Interpolate Event search filter;
2 means the Interpolate replace filter.

'**range**' is one of the following:

0 Note Key;
1 Note Velocity;
2 Note Duration ;
3 Key After touch Key;
4 Key After touch Pressure (value);
5 Controller Number;
6 Controller Value;
7 Patch Number;
8 Channel After touch Pressure (value);
9 Wheel Amount;
10 Channel;
11 Beat;
12 Tick;
13 Patch bank number (new in 6.0);
14 14= RPN Number (new in 6.0);
15 RPN Value (new in 6.0);
16 NRPN Number (new in 6.0);

17 NRPN Value (new in 6.0).

'**inrange**' is the opposite of the '**Not checkbox**':

"FALSE" means '**Not**' is checked;

"TRUE" means it is not checked.

'**min**', '**max**' are the minimum and maximum values for the range.

2.4.13.2 FILE Menu Functions

`FileExtract [<pathname>])`

If '**pathname**' argument is omitted, then the user is prompted for the name.

If '**pathname**' already exists, then user is warned and given the ability to enter a different name.

Returns: "FALSE" if file was not saved. This could be either because of an error writing the file or because the user pressed '**Cancel**' in the dialog.

`(FileMerge [<pathname>])`

If '**pathname**' argument is omitted, then the user is prompted for the name.

Returns: "FALSE" if file could not be loaded. This could be either because of an error reading the file or because the user pressed '**Cancel**' in the dialog.

`(FileNew [<template>])`

If '**template**' argument is omitted, then the user is prompted for the name of a template. The '**template**' argument is a string and should contain only the base name of the template, without an extension. For example '**Normal**' is correct but '**Normal.TPL**' is not.

Returns: "FALSE" if the template was not found and loaded. This could be either because of an error reading the file or because the user pressed '**Cancel**' in the dialog. The '**Normal**' template always works: if no '**Normal.TPL**' file is found, a blank work file is created with the normal program defaults.

`(FileOpen [<pathname>])`

If '**pathname**' argument is omitted, then the user is prompted for the name.

Returns: "FALSE" if file was not loaded. This could be either because of an error reading the file or because the user pressed '**Cancel**' in the dialog.

`(FileSave)`

Returns: "FALSE" if file was not saved due to an error writing the file.

`(FileSaveAs [<pathname>])`

If '**pathname**' argument is omitted, then the user is prompted for the name.

If '**pathname**' already exists, then user is warned and given the ability to enter a different name.

Returns: "FALSE" if file was not saved. This could be either because of an error writing the file or because the user pressed '**Cancel**' in the dialog.

2.4.13.3 GOTO Menu Functions

(GotoSearch [<noPrompt>])

The '**noPrompt**' argument is simply a syntactic placeholder. It can be anything. If there is any argument, the user will not be prompted with the Search dialog. Instead, Cakewalk will perform the search using the Search Event Filter as-is.

When Cakewalk records this command for you into the CAL editor, it first uses one or more calls to the "**ResetFilter**", "**SetFilterKind**", and "**SetFilterRange**" functions, all using the filter number 1. Then it adds:

```
(GotoSearch 1)
```

The dummy argument '**1**' means not to prompt the user.

If you are writing a CAL program where you want the user to be prompted for completion of the Search dialog, then call "**GotoSearch**" with no arguments:

```
(GotoSearch)
```

(GotoSearchNext)

Repeats the last search.

2.4.13.4 SETTINGS Menu Functions

(SettingsChannelTable [<on> <n1> <n2> <n3> <n4> <n5> <n6> <n7> <n8> <n9> <n10> <n11> <n12> <n13> <n14> <n15> <n16>])

n1 through n16 are track numbers, 0..255.

(SettingsMetronome [<play> <rec> <acc> <count> <port> <chan> <key> <vel> <dur> <beep>])

(SettingsMidiIn [<n1> <n2> <n3> <n4> <n5> <n6> <n7> <n8> <n9> <n10> <n11> <n12> <n13> <n14> <n15> <n16>])

n1 through n16 are MIDI channels. A value of zero means the channel is not recorded or echoed. A value of one means that it is recorded or echoed.

(SettingsMidiOut [<txmidirt> <sendcont> <sendspp> <sppdelay> <ctrlzero> <ctrlchase>])

These arguments correspond to the similarly-named items in the MIDI Out dialog box.

(SettingsMidiThru [<mode> <port> <chan> <key> <vel> <localonport>])

These arguments correspond to the similarly-named items in the MIDI Thru dialog box.

(SettingsRecordFilter [<note> <keyaft> <control> <patch> <chanaft> <wheel>])

These arguments correspond to the similarly-named items in the Record Filter dialog box.

2.4.13.5 TRACK Menu Functions

In all the mentioned track menu functions the value of the parameter '**track**' must be between 0 and 255, unless otherwise stated..

(TrackArchive [<archive> [<track>]…])

This function handles the track archive-status. An **archived** track is silenced during playback. You must stop playback to change a track's '**archive**' status.

'**archive**' must be one of the following:

1	archive;
0	un-archive;
-1	toggle.

(TrackActive [<active> [<track>]])

This function sets the track active or muted. However, you can turn a muted track '**active**' during playback.

'**active**' must be one of the following:

1	playing;
0	mute;
-1	toggle.

(TrackBank [<bank> [<track>]])

This functions sets the track bank.

'**bank**' must be -1..16384.

(TrackChannel[31] [<chan> [<track>]])

This function sets the track channel.

'**chan**' must be 0..15.

(TrackKey+ [<amount> [<track>]])

This function sets how much numbers of semi-tones the track will be transposed.

'**amount**' must be -127..127.

(TrackName <name> <track>)

This function sets the '**name**' of a '**track**'. The '**name**' is either a string literal or a string created by a function.

(TrackPan [<pan> [<track>]])

This function sets the panning of the '**track**'.

'**pan**' must be -1..127.

(TrackPatch [<patch> [<track>]])

This function sets the '**patch**' of the '**track**'.

'**patch**' must be -1..127.

[31] the function was called: TrackChan in CAL version 3.

(TrackPort [<port> [<track>]])

This function sets the output '**port**' of the '**track**'.

'**port'** must be 0..15.

(TrackSelect <state> <track>)[32]

This function defines whether a track is selected or not selected. '**state**' must be one of the following:

1	select;
0	un-select;
-1	toggle.

'**track**' be -1 to mean '**all tracks**', otherwise it must be from 0 to 255.

Note: This only has an effect on old Cakewalk 3.0 compatible CAL functions. For example, this will change which tracks '**EditCut**' operates on, but not '**EditCut40**'. The latter strictly follows what the user has selected. See also chapter 5.1.1: 'Downwards compatibility'.

(TrackTime+ [<ticks> [<track>]])

Sets an offset (Time+) to the start time of the events in the '**track** . Is does not affect the stored time for each event in the '**track**'.

The '**raw**' time as a result of the value '**ticks**' must be between 0 and the maximum value of '**raw**' time.

(TrackVel+ [<amount> [<track>]])

This function sets an offset (Vel+) to the velocity of every note in a '**track**'.

'**amount**' must be between -127..127.

(TrackVolume [<volume> [<track>]])

This function sets the '**track**' volume.

'**volume**' must be between -1..127.

[32] SONAR: To be able to be selected tracks must contain events.

3 Creating your CAL programs

Before SONAR, Cakewalk had a built-in editor for CAL programs. With SONAR you need to use an ASCII editor to be able to create CAL programs.

It is not clear to me whether this is the first step of 'Twelve Tone Systems' in killing CAL in later versions of CAKEWALK. I believe doing so will be a big mistake, because CAL is one of the features, which make the difference with competitive the products.

When you use a **standard** editor or a word processor to compose CAL programs, use the following settings. Set TAB spacing to .15 inch, set the page left and right margins to .25 each and if you have the option, select an even-spaced font instead of a proportional font. The CAL editor used an even spaced font so doing the same in your alternate editor allows you to see closer to the final result this way. Save your file as '**plain text**' or '**ASCII text**' depending on how it is worded. Microsoft Word 2000 for instance, will normally add the '**.TXT**' to the file name. You can force a '**.CAL**' extension to the file by entering the filename – including the extension CAL – within double quotes. E.g. use 'Save As', select file type TXT, and enter '**Program name.cal**'. Program name is the name of your CAL program.

Other editors which can be used without the above-mentioned solution for MS Word, are: Notepad in Windows (Microsoft) and WordPro (Lotus).

4 Programming techniques

I will here present standard software programming conventions and techniques. They are not unique, but essential in making programs, which can be maintained, and understood also by other programmers. Also you need to realize that after a couple of years many software programmers do not understand their own programs anymore.

All here provided sample programs can be found on http://www.MIDI-Kit.nl. Unless otherwise stated the examples can be found in a ZIP file called: "CAL program samples.zip".

The program examples will help you to understand the programming rules of CAL.

4.1 Conventions

4.1.1 Program header

The Header of a program must provide the following in formation:

- Program name;
- Copyright statement;
- Version, with date of creation, and if the version is a revision of an earlier version: an explanation of the changes;
- Purpose of the program;
- Required user actions, e.g. selection of the to be modified events;
- Input parameters.

Let us look at a simple program that displays the version of CAL:

```
;  Show version.cal
;  (c) Copyright 2004 T. Valkenburgh
;  Version 1.1, August 2004:   Check whether version is 2.0 or higher
;  Version 1.0, May 2004
;
; This routine shows the CAL version.
; Input parameters:      None
;
(do
     (include "need20.cal")     ; Require version 2.0 or higher of CAL
;Get the version and display it
     (pause "The version of CAL is: " (/  VERSION  10) "."( %  VERSION  10))
) ; end of do
; end of program
```

Notice that this program of two statement starts with the "do" function; it contains more than one statement. No declarations are needed; no variables are used.

Run this program to check the version of CAL. In SONAR 3 you will get 100. For a presentation in the format: *version 10.0*, you need to divide "VERSION" by 10 and use the remainder of '"VERSION" divided by 10' after the dot.

The first function is a check on the CAL version. Do always include such a test. E.g. you may have decided that you guarantee your programs only for CAL levels 10.0 and higher. However, you want to make it available via Internet. It is then key to include a test on CAL version 10.0. The '**Need20.cal**' program – shipped with Cakewalk – can easily be changed into a '**Need100.cal**' program. A more generic solution is given in chapter 4.7 : 'Building an include library'.

The "**if**" function may be difficult to read. However, you can make it more readable in the following way:

```
if (== Event.Kind NOTE)
      ; then
            (some function)
      ; else
            (another function)
) ; end of if
```

To get a good picture of the nestled functions, I always add to the closing parenthesis a comment like:

```
) ; end of do
) ; end of while
```

Sometimes I even tell in the comment the condition of e.g. the while function.

I also have the habit to end my programs with:

```
; end of program
```

The reason is that if you have a program that exceeds more than one page, you are sure that the last page on your desk is really the last page of your program. In all below shown examples, we will use this convention. In 'Program skeleton.cal' (chapter 7: 'Examples') you can find the base for creating your CAL programs.

4.1.2 Variable names

Variable names must be meaningful and easy to remember.

To ensure that the name I use, cannot be or become - in later versions - an internal name of CAL, I always use a lowercase character indicating the type of variable.

```
nChannel
```

This variable is the channel number; therefore, it begins with 'n'.

Below I give my convention for the type indication:

Type	First character
Boolean	b
Number	n
Raw time	r
Text	t

Type	First character

4.2 Error handling

It often has been said that 20 % of a program is functional code, and 80 % is code for handling exceptional situations. This indicates how important error handling is. Especially if you want to make your program available to other users, it is important to test on exceptional situations, and provide clear error messages.

Not checking whether the user has provided the expected input may result in an hang up in CAKEWALK.

If the user must select “Events” to be able to run your program, then you should test whether “Events” are selected. Out of range values of velocity, pitch, etc. can give very unexpected results. It is key to include these tests in your program to avoid exceptional situations.

4.2.1 Checking the marking of Events

Sometimes programs hang if no “Events” have been selected (marked). Therefore, you must test whether “Events” are selected. A simple program can be used for the test. I have written a generic include program for the test. This program will count the events, and return in a variable the number of events. The calling program provides the kind of event in the variable ‘**nEvent**‘. Let us look at this include program[33].

```
;
; '+Events marked.cal'
;
; (C) Copyright 2004 T. Valkenburgh
; Version 1.0, October 2004
;
; Purpose
; Include program that counts the number of marked Events.
; Events with the same Event.Time in the same track are counted as ONE.
;
; Input parameters:      nEvent       Event.Kind to be checked
; Output:                nEventCount  Number of events
;
(do
; Variable declarations of locals
     (dword ~rPrevious 0)                    ; Previous Event.Time
; Set Event count zero, just to be sure
     (= nEventCount 0)
; search for Events
```

[33] For an explanation of the naming convention of the variables see chapter 4.7: ‘Building an include library’.

```
    (forEachEvent
          (if (== Event.Kind nEvent)        ; nEvent has been provided by
                                            ;  the calling program
                (do
                      (if (&& (== Event.Time 0) (== nEventCount 0))
                            ; then
                            (= nEventCount 1) ; first event on time zero
                            ; else
                            (if (!= Event.Time ~rPrevious)
                                  (do
                                        (++ nEventCount) ; count
                                        (= ~rPrevious Event.Time)
                                  ) ; end of do
                            ) ; end of if
                      ) ; end of if
                ) ; end of do
          ) ; end of if
    ) ; end of forEachEvent
    (undef ~rPrevious)                      ; remove definition
) ; end of do
; end of program
```

"Events" at the same "Event.Time" in the same track are counted as one. This allows you to see a chord as one event. The calling program can test whether the count is zero. This will look like:

```
(int nEvent NOTE)               ; Event for checking whether they are marked
; Get the number of marked notes
    (include "+Events marked.cal")
; Check whether notes has been marked
    (if (== nEventCount 0)
          (do
                (pause "No note marked, process terminated")
                (exit)
          ) ; end of do
    ) ; end of if
```

The program '**+Events marked.cal**' is part of the include library. See also chapter 4.7: 'Building an include library'.

4.2.2 Preventing over-range Problems

Something very odd happens when you send some event parameters negative or over their allowed limits. The effect is referred to in computer circles as **aliasing**. What this means is the value assigned to the variable will be something that appears unrelated to the actual value that should have been stored there. For example, suppose in the act of scaling a note velocity, the value goes

negative. This negative value is then entered into the sequence as the note's velocity even though it's impossible for a note to have a negative velocity. Now you look at that note in the '**Event**' view and notice that its value is indeed a positive number, but seemingly unrelated to the results of the equation that generated it. The value has been '**aliased**'. Because the '**pocket**' for a velocity value cannot hold a negative number, the '**negative sign**' of the value has been stripped away and the remaining bits that made up the negative number were left behind as a bogus positive number. Even though most of the time attempting to place a number that is too large for a variable will result in an error message, driving a result negative will often go unnoticed by CAL. To prevent aliasing and over range error messages, it is important to add over range checking to any user input and calculation that can result in a variable going over or under allowable limits.

4.2.2.1 Input range checking

You must always realize that external and internal values can be different. E.g. Channel numbers range 1 through 16 externally, but are internally presented as 0 through 15.

That means you ask the user to input a channel between 1 and 15, ensures checking the range, and converts the value for internal use by decreasing it with 1.

In your input statement you set the input range that is allowed for the field you ask the user to provide:

```
(getInt  nChannel "Channel  = " 1 16)
(--  nChannel)                          ; internally CAL uses 0 … 15
```

This will ensure that the channel will be within a valid range, and that you will use the correct internal CAL value.

4.2.2.2 Preventing internal over-range errors

If you calculate values e.g. key velocity, you must ensure that the result within a valid range, e.g. between 0 and 127 for velocities. Below I show you what I mean. Let's say we're scaling velocities. At the end of the scaling formula simply add:

```
(some calculation that scales velocities)
(if (< velocity 0) (= velocity 0))          ;Correct negative velocities.
(if (> velocity 127) (= velocity 127))      ;Correct velocities that overrange.
(and go on with your program)
```

These two extra lines of code test for, in the first, the variable '**velocity**' being less than zero, in other words, a negative number. If this is found to be so, '**velocity**' is set equal to zero. In the second, if '**velocity**' is greater than 127, then it is set equal to 127. This way, no matter what result your scaling formula generates, the value ultimately assigned to '**velocity**' will be within the allowable range for that parameter. You can of course also add a warning message, to ensure the user is noticed that an over range situation occurred.

4.3 Mathematics

Calculations can give unexpected results within CAL, because CAL does not have floating point calculations. All calculations within CAL are done with whole numbers, and that means you must be very carefully with creating formulas within CAL. Let me show with an example how your program can give different results, while the formulas in both cases are correct. In the example below both calculations give the same result.

```
5 * 12 / 10 = 6
```

```
5 / 10 * 12 = 6
```

Let us now look at the two CAL programs:

```
(/ (* 5 12) 10)
```

This example gives as result: **6**.

```
(* (/ 5 10) 12)
```

This example gives as result: **0**.

The reason is that 5 divided by 10 gives in CAL 0 with 5 as remainder. If we do not want to expand our calculation with using the remainder, we will get the wrong answer.

In most cases the remainder is not important for CAL programs. You can avoid this problem by doing the multiplication before the division. Make sure that the variables can hold the largest value during the calculations. If needed, use a "dword" or "long" variable.

4.4 Sending SYSX messages

Here I will describe how to send SYSX messages to the synthesizer. I will use an example program that switches the General MIDI mode of your synthesizer. The SYSX message for switching the GM mode of your synthesizer is:

```
F0[34]  ; indentifies System Exclusive message start
7E   ; ID number: Universial Non-realtime message
7F   ; Device ID: Broadcast[35]
9     ; Sub ID number 1: General MIDI Broadcast[36]
xx   ; SUB ID number 2: indicates GM mode( GM1=1, GM2=3, GM off=2)
F7   ; EOX : end of exclusive
```

With the CAL function "**sendMIDI**" we can send the SYSX message. CAL can only handle decimal numbers. Therefore, we must convert the hexadecimal numbers into decimal.

The user will be asked for selection of: GM1, GM 2 of off. We assume that the synthesizer is connected on port 1, which is true in most cases. Further this small program is clear enough.

```
; 'GM Mode.cal'
;
; (C) Copyright 2005 T. Valkenburgh
; Version 1.0, February 2005
;
; Purpose
; Switching between GM modes of a MIDI device.
;
; Input parameters:      'GM Mode (GM 1=1, GM 2=2, Off=0)'
; Output:          General MIDI message to synthesizer
;
(do
; Variable declarations
```

[34] All numbers are in hexadecimal format.
[35] This must be according to the specific instrument. See the owners manual of the instrument.
[36] This must be according to the specific instrument. See the owners manual of the instrument.

```
     (int nCALVersion 31)              ; required CAL version
; Test on CAL version
     (include "+need version.cal")     ; terminate if version is too low
;
     (int nPort 0)                     ; assumes your synthesizer is on port 1
     (int nChannel 15)          ; any number is ok, it will be ignored
     (int nMode 1)              ; Mode
; SYSX message contents
     (int nIDnumber 126)        ; Universial Non-realtime Message (=7EH)
     (int nDeviceID 127)        ; Broadcast (=7FH)
     (int nSubID1 9)            ; General MIDI message (=09H)
     (int nSubID2 1)            ; General MIDI 1 On (=01H)
;
; Program
     (getInt nMode "GM Mode (GM 1=1, GM 2=2, Off=0) " 0 2)
     (switch nMode
           0      (= nSubID2 2); GM 2
           2      (= nSubID2 3); GM off
     ) ; end of switch
; send General MIDI message
     (sendMIDI nPort nChannel SYSX  nIDnumber nDeviceID nSubID1 nSubID2)
) ; end of do
; end of program
```

4.5 Switch Tree Menus

When I introduced the "**switch**" function in chapter 2.4.10: Control flow functions, I touched on the idea of using a "**switch**" tree to control a menu. The execution of this concept is very simple. Look at this example:

```
(int mode 0)
(getInt mode "Select Program You Wish To Run  " 0 3)
(switch mode
   0
        (do
            (a bunch of stuff that makes up a CAL program)
         ) ; end of do
   1
         (do
            (another program the user could run)
          ) ; end of do
   2
       (do
           (you get the idea)
```

```
        ) ; end of do
    mode
        (do
           (the last of the program selections)
        ) ; end of do
) ; end of switch
```

In this example, we have used "**switch**" as a menu driver. The user selects which of four CAL programs they wish to run and the tree picks out the right one based on the value of '**mode**'.

However, what about this example:

```
(forEachEvent
   (if (== Event.Kind NOTE)
      (switch Note.Key
         35
            (+ Note.Key 1)
         38
            (+ Note.Key 2)
         49
            (+ Note.Key 8)
         Note.Key
            (- Note.Vel 10)
    ) ; end of switch Note.Key
) ; end of forEachEvent
```

Suppose we have a general MIDI percussion track selected. Here we start a "**forEachEvent**" loop and filter out everything except note events using the "**if**". Now we set up a "**switch**" tree that performs a special change on certain notes, in this case it changes Acoustic Bass drum notes (note number 35) to Bass Drum #1 notes by adding a 1 to the note number. If the loop finds an Acoustic Snare note (#38), it adds 2 to it and it becomes an Electric Snare note. Any Crash Cymbal #1 notes (#49) it finds are changed to Crash Cymbal #2 notes, and everything else, including native Bass Drum #1, Electric Snare and Crash Cymbal #2 notes are lowered in velocity by 10 units to make the changed notes stand out a bit. As you can see, in this case we needed an '**if all else fails**' trap so we don't have to list a case for every single note in the drum kit.

4.6 Implementing On-Line help in a CAL program

As odd as it may seem, there is a way to provide the user of your CAL programs a set of simple instructions that they can look at or not as they feel necessary. This can be done through the use if a "**while**" loop enclosing a group of "**pause**" dialogs. One very useful reason to employ this technique is due to the implementation of presets in a program. In this way, the user can choose from one of several presets that auto-configure the program for specific operations, choose to enter all parameters by hand or choose to view a series of '**Help**' boxes that describe the presets and offer other pertinent information. Here is an example of some code that will generate the help boxes.

```
(int preset -1)                    ;Declare the preset variable.
 (while (== preset -1)             ;So long as preset  equals -1...
  (do                              ;do the following, starting with a request
; for user input.
    (getInt preset "Select Preset Mode or Hit ENTER to review a list of presets "  -1 7)
```

```
    (if (== preset -1)              ;If the default value is still in "preset"...
     (do                            ;display the following "pause" boxes.
      (pause "Note Qualification Preset List : 0=Set All Variables By Hand 1=Spot Apply")
      (pause "MONOPHONIC Track preset 2=Quarter Notes or Bigger 3=All Long Notes")
      (pause "4=Only First Note in a Phrase 5=Only Last Note in a Phrase")
        (pause "POLYPHONIC Track presets 6=All Notes 7=Only Last Note of Runs")
       ) ; end of do,  nesting the "pause" statements.
     ) ; end of  if
  ) ; end of do
) ; end of while
```

In the first line, we declare a variable that we will use as our preset selection variable. It is initialized to -1 so that in the input dialog that follows, the user can simply hit ENTER to select this default value and thus read the '**Help**' boxes.

After declaring '**preset**', we start a "**while**" loop that continues to loop for as long as '**preset**' is -1. Next is that input dialog where the user can enter a preset number from 0 to 7 or keep the default value of -1 and read the '**Help**'. The "**if**" function triggers if the value of '**preset**' is still -1. If is isn't, then the "**if**" is bypassed and we return to the top of the "**while**" loop where the condition for the loop fails and execution immediately proceeds with the line of code after the "**while**" loop's closing parenthesis. If '**preset**' is still -1, then the "**if**" condition passes and we execute its nested functions which consist of a series of "**pause**" statements containing some useful information in their text arguments. The user reads this information, and from it decides which preset they wish to select. After the last "**pause**" dialog box, the "**if**" exits and we come back to the top of the "**while**" loop.

The variable '**preset**' still equals -1, so we start the loop again. This time, the user will decide on a preset based upon what they read in the "**pause**" boxes. The value is entered during the "**getInt**" statement. The "**if**" now fails because '**preset**' has been set to a value other than -1. The "**while**" loop likewise fails for the same reason and the program continues, likely with a "**switch**" tree to process the user's selection in the '**preset**' variable.

4.7 Building an include library

If you are going to use CAL a lot, it might be a good idea to look into constructing a library of CAL modules that perform specific '**micro**' tasks and use them as building blocks to make larger CAL programs. You can also build slightly different versions of a basic program that each performs the same task but with different preset parameters or focus.

There is , however, an important consideration if you use this function within a loop like "**forEachEvent**" or "**while**". This is because each time the loop is run, CAL has to go searching for the included program, load it and run it. Even with the included program residing in the disk cache memory, this can be a very time consuming process and will slow down execution of your parent CAL program considerably. Loops in the included programs can contain such loops without degradation of performance.

If you are going to make this idea work, you must be organized about it. Decide on some conventions and stick to them.

One convention might be to make sure all local variables your "**include**" programs declare are named fundamentally different from the way you name variables in parent programs. This will prevent any 'Variable redefined' erors from popping up at run time.

If the parent program needs pro pass parameters to the "**include**" program, they must share variables. To ensure general use of include programs require a very strict naming convention to

avoid problems.

In addition, you should use the "**undef**" function to **'undefine**' the local variables used within the include program, and not used to pass through to the parent program. This will prevent unexpected interaction between "**include**" programs.

It will help if the "**include**" programs can easy be recognized in your folders, therefore, also here a good naming convention helps.

All these aspects of passing data back and forth between parent programs and their included child programs should follow a set of protocols. Keep names of variables consistent by function so that as you write a new parent, it will easily dovetail into your current scheme. Use the same structure for assembling data to be passed to any included program and collecting returned data. Document in the header the input and output parameters.

There is one other consideration worth mentioning. An "**include**" function will use up a nesting level on its own, not to mention any nesting levels created by the included program. CAL can only handle a limited[37] nesting levels before issuing an 'Evaluation stack overflow' error. Keep included programs very efficient and don't run "**include**" functions if they themselves will be nested very deep within a function of a parent program.

Herewith my set of library conventions:

Item	Rule
Name of "**include**" program	Name starts with +
Local variable names in "**include**" program	Name start with ~

Below I show how the program **'+need version.cal**' will look like with these conventions.

```
; +need version.cal
; (C) 2004 T. Valkenburgh
; Courtesy: this program is based on the by Twelve Tone Systems
;             provided 'need20.cal'
; Version 1.0, August 2004
; This library (include) program checks whether the version of CAL is
; equal or higher than the level indicated in the input parameter
;
; input parameters:         nCALVersion  level on which must be tested
; output parameters:        none
;
(if (< VERSION nCALVersion)
 (do
   (pause "This program requires CAL version " (/ nCALVersion 10) "." (% nCALVersion 10))
   (exit)
 ) ; end of do
) ; end of if
; end of program
```

Remarks:

- Because I use program code from somebody else, I mention that in my header;

[37] The number of allowed nested "**include**" programs is not specified by 'Twelve Tone Systems".

- The calling program provides the input parameter – required level of CAL - via a variable '**nCALVersion**'. The program tests and returns if ok, or exits with an appropriate message to the user if the level is lower than the input parameter. Also note that the program does not start with the "**do**" function. The "**if**" statement includes all other statements;
- This include program has no local variables.

5 Tips, and work-arounds

Not everything about CAL is straightforward. There are allot of aspects of CAL that are undocumented and in some cases, wrong documented. There are subtle features to some functions that don't present themselves on the surface. You can do things with CAL that wouldn't occur to you just off-hand. Then again there are the quirks in CAL that make you want to pull your hair out! In an attempt to forestall unnecessary baldness among CAL users, I wish to provide you with some of the fruits of my own hair pulling sessions with CAL. Keep in mind that there must be at least as much that I have yet to discover. Perhaps with the collective efforts of all of us CAL programmers, we can keep each other sane.

Here is my opening contribution to this cause. One thing I wish to convey as a sort of disclaimer is that you may never see some of these reports of strange behavior. Some of them I myself have only seen once or twice and perhaps they are a result of some odd conflicts in the registry or whatever. I still feel compelled to mention them JUST IN CASE someone else out there comes up against the same problem. This way, they will have at least been briefed on the possibility and can take action accordingly. Fact is, I hope nobody ever runs across the oddities that are listed here, but you will at least some of them, believe me!

Always remember CAL programs can be up to 32768 characters long – including comments. I do not know whether you can solve this by putting the code in separate programs and include these programs in your main program. Until now, I never reached this limit.

5.1 CAL Compatibility

Compatibility can be upwards and downwards.

Downwards compatibility is easier, because you can test with all previous versions – if available - of CAL. However, making your program downwards compatible may limit you to the lowest functionality. Also you may discover that certain functions work out differently in the various levels of CAL.

Upwards compatibility is more difficult to guarantee. However, by following a strict discipline you can increase the compatibility. Whether you succeed or not succeed is than more dependent on the quality of the CAL developers. History has shown that especially the menu functions may be incompatible due to additional functionality.

5.1.1 Downwards compatibility

By and large, anything you write with CAL on a system running Cakewalk version 3 should load and run fine on any version after it. The same applies to stepping up from version 4, 5, 6 etc. to any higher version. However (and this should come as no surprise), the reverse is not true. Not only are some functions different or even unavailable in earlier versions, but also the system itself can impose limitations.

- Something to keep in mind; if you are writing code that will be using event constants that are only recognized by later versions of CAL, and you want the code to be backwards compatible with earlier versions, you cannot reference these constants by name but by the number they equal. If you use the name, the versions of CAL that don't support them will give you an error message and refuse to run. On the other hand, if you use the number, then CAL will simply use this number and dismiss the result without issuing an error. For example, suppose you wish to hunt for RPN events in a program that otherwise is just as useful in Cakewalk versions other than 6 or above. If you use a function like:

```
(if (== Event.Kind RPN)
```

CAL versions below 6 will go belly-up. On the other hand, if you write it like this:

```
(if (== Event.Kind 9)
```

CAL versions 6 and above will still respond "TRUE" if they find an "RPN" event. Other versions of CAL may simply return results during the comparison that reflect they were either unsuccessful in finding an event that doesn't exist in any sequence that could be loaded into that version of Cakewalk, or they found something they don't know by name but never the less spotted by value.

However, realize that the developers of CAL can change the internal values in later versions. It is a good programming practice **to use constants only by name** and not by value. This will ensure upwards compatibility of your programs.

- One of the most obvious came to Glen when he was working with one of his CAL programs called '**Crossfade.cal**'. He wrote it in version 3 and had to place some limitations on it due to system constraints.

  After reworking it in version 7, it no longer ran in version 3, not because he had added features not supported by version 3, but because it overran the abilities of the version 3 run-time interpreter.

  He added two more nesting levels to the program, and then he added the ability to select '**none**' for the amount of attack and/or decay time in the cross fade of each note.

  On version 7, this was a great new feature but in version 3 it generated an 'Evaluation stack overflow' error due to the addition of the new and deeper nesting of some functions. His indention's had gone from 8 to 10 tabs deep in the code text and version 3 simply can't keep track of that many calls.

  In that he realized he would have to have separate versions of '**Crossfade.cal**' for version 3. He also included in the new version the ability to deselect the target track and select a new target track for performing the volume event integration on the cloned area, a feature impossible in version 3 because there is no way to deselect the **blue-highlighted** track.

  Another approach Glen could have taken was to build in his program a test on the CAL version, and based on the outcome of the test e.g. include a program variant for that specific CAL version.

- Herewith another example: in later Cakewalk versions you can deselect all tracks and select new ones through CAL even though if you were to read the CAL help in these later versions it will try to convince you that the "**TrackSelect**" function doesn't work any more - **wrong**!!! It works just fine. There are some limitations to it now, like the fact that some functions, the "**insert**" function for one, will not honor the ("**TrackSelect**") function unless it is within a ("**forEachEvent**") loop. If you try to use ("**TrackSelect**") and it doesn't work, you may have just run into this limitation.

- Another consideration is a minor kink in the CAL execution system in versions 7 and 8. If you load a CAL program or otherwise attempt to use CAL without a '**.WRK**' file loaded into the workspace, there is no way of running the CAL program. Even clicking on the "**Run**" icon in CAL View will have no effect at all. The only time this is a problem if you have cleared the workspace and then want to load some files or format some default workspace to your liking in one step by running a CAL program that pre configures everything just so. You can forget it! You must load **something** into the workspace first, even if it's just a default blank workspace. Now you can run your CAL set-up program. This isn't a problem when you first load Cakewalk as you are given a default blank workspace as soon as the program finishes loading. It is only a problem if you have closed everything and the workspace is completely vacant.

- As a quick aside, some functions – as mentioned above - such as the new, advanced '**Edit**' menu functions cannot run on a Cakewalk version 3 system, and some of them may not run on

versions 4 or 5 if they contain version 6 arguments. Keep this in mind, and if your program requires a more advanced CAL system than 2.0, you will have to call similar programs to test for "VERSION" being less than 31, 40 or 60 depending on the point where you wish the test to fail.

At the very start of a CAL program, it's a good idea to "include" the small '+need version.cal' program – see chapter 4.7: 'Building an include library' - that tests the version level of the CAL system the user is running and makes sure that it is sufficient for running the program. If the test in '+need version.cal' fails, the entire CAL system aborts and all execution stops.

5.1.2 Upwards compatibility

- To ensure upwards compatibility you must always use internal CAL names instead of the values. E.g. never use '144' instead of "NOTE", because the developers of CAL may decide to change internal numbers.
- Another aspect is, try to understand the CAL functions very thoroughly. If you try to understand functions by experimenting with trial and error, you may end in using a bug instead of a meant functionality. If the CAL developers fix this bug later, your program will not work anymore.

Unfortunately CAL is so badly documented that sometimes you need the trial and error approach.

This does not mean that you will not have any problems when you upgrade to a higher level of Cakewalk. Let me give an example:

In Cakewalk Pro Audio when you mark one note, the "From" and "Thru" variables are equal, with exception of the last note in the sequence. With the last marked note the difference between the "Thru" and "From" variables is the note duration.

In Cakewalk SONAR the difference between the "Thru" and "From" variables is always the note duration.

5.2 Details of event sequences

- If you assign a MIDI channel to a track, events on that track will be broadcast over that channel even if the event has a different channel assignment within its data pattern. This is why you can bounce events from track 2 on channel 2 to track 4 on channel 4 and have them all go out on channel 4.
  You can see this oddity if you examine a track by selecting the '**Event List**' view and look at events that were bounced from another track. They will likely be allocated to a different channel than the events native to the track. However, all of them will output to the channel for that track providing one has been selected. If you were to select none (**--**) for the track channel by inserting a '**0**', then the events would be broadcast on whatever channel their event data says.
  This is good to know because events on a track can be segregated for editing by changing their MIDI channel without really changing the channel they will be broadcast to. Be aware that even though we call the MIDI channels 1 through 16, CAL internally calls them channels 0 through 15.

- Although it may not seem important on the surface, different events have different amounts of data associated with them. While all events have a time, at which they are to be broadcast, and a channel they are set to broadcast over. Note events will also have a pitch, velocity and duration as part of their data content or a controller event will have a controller number and a controller value. All data associated with an event will be referred to as one or more of its '**variables**', because these values can be read and changed in the editing process. When writing

CAL programs, one of the easiest mistakes to make is to forget what event type has what kinds of variables.

- A note event has a duration variable associated with it. As a result, when the note is sent over MIDI, a '**Note On**' command will be sent at the start-time of the note, and a separate '**Note Off**' command (usually another '**Note On**' command but with a velocity of '**0**') is sent after the duration time has elapsed. Therefore, even though the note is displayed as one event in the sequence, it results in the broadcasting of two separate MIDI commands in order to satisfy its purpose. If you have ever '**Pasted with replace**' a measure over a note that extended from the previous measure into the one being pasted over, you may have experienced a stuck note due to the confusion that can be caused by overwriting a sustained '**Note Off**'.

5.3 Correcting sequences which do not look sequential

SONAR supports the concept of '**Clips**' within a track. '**Linked Clips**' can be used to create repetitive '**Clips**'. The use of '**Linked Clips**' across tracks, however, may be seen by your CAL program as **out of sequence**.

If your CAL program is sensitive to the sequence of the "Events"events you must combine the clips in one track. For further information the SONAR manual and help files.

5.4 Forcing Version 6 (EditInterpolate) To Use Markers

As mentioned several times elsewhere, the "**EditInterpolate**" function doesn't work correctly in version 6. In other versions, it responds to the "From" and "Thru" marker values without any problem, operating only on the selected portion of the sequence defined by these markers. In version 6, "**EditInterpolate**" will not respond to the markers and will operate on an entire track from beginning to end regardless of the values in "From" and "Thru".

As it turns out, it is possible to switch the "**EditInterpolate**" function's response to the "From" and "Thru" markers back on. Glen discovered this quite by accident one day and this is how Glen did it. Just before the function statement, you add a pair of **dummy** statements that don't have any use as CAL operations, but somehow forces the "**EditInterpolate**" function that follows to use the markers. Just type in these two lines above the (EditInterpolate 1) function as shown:

```
(= From From)

(= Thru Thru)

(EditInterpolate 1)
```

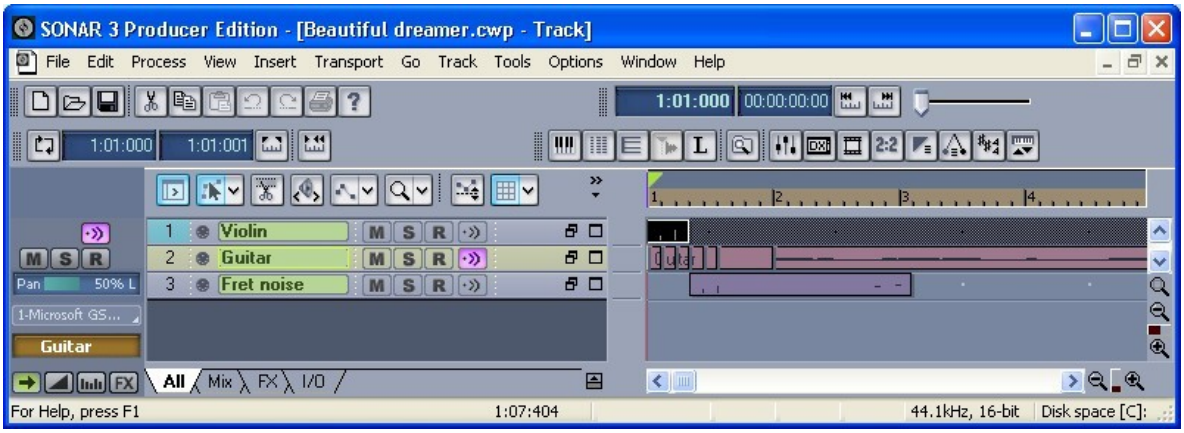
Only now will CAL honor the values in these markers. This can lead to some real confusion in writing CAL programs using "**EditInterpolate**" if you aren't aware of this oddity which is, even more oddly, completely undocumented. Also, keep in mind that you must do this just before every single "**EditInterpolate**" function in your program, as one instance of these **dummy** statements isn't enough to condition all of the interpolation functions in a program. For convenience, if you are writing code that needs to be portable across several versions of Cakewalk, just make it a practice of writing "**EditInterpolate**" functions as mini macros of three lines; the two dummy statements and the interpolate statement. In other versions of CAL, the extra statements will have no effect, neither good nor bad. In version 6, they will save your butt!

5.5 Explicit and implicit track selection

From version 3 to SONAR, there have been changes along the way in the representation of selected tracks. In Cakewalk you have explicit and implicit selected tracks. This can sometimes create confusion. In every version, one track is always implicitly selected regardless of any other action.

The behaviour of Cakewalk with regard to explicit and implicit selected track can create confusion. In the picture below you can see the different track situations for SONAR:

- Track 1: an explicit selected track;
- Track 2: an implicit selected track;
- Track 3: a not selected track.



In the various Cakewalk versions the tracks may be identified differently from SONAR.

Explicitly selected tracks are shown by their track numbers highlighted in blue if you are using the default colors.

The important thing about an implicitly selected track is, that with no explicitly selected tracks available, the implicitly selected track will be the default focus of many actions that the user can take including, for example, opening the '**Event List**' view. On the other hand, explicitly selected tracks are the only ones subject to the user's editing actions and most of the operations of CAL programs.

If you perform an edit or run a CAL program in Cakewalk version 3, the implicitly selected track will always be affected along with any other tracks that might be explicitly selected. The drag is that although any explicitly selected track can be selected and likewise deselected at will, either by clicking on the track or by executing the ("**TrackSelect**") function from CAL, the implicitly selected track cannot be deselected, only changed from one track to another by the user. Unfortunately, it is equally subject to any actions about to befall the explicitly selected tracks. It impossible to have **no** selected track in Cakewalk 3. This is not true for the other versions. In versions 4 till SONAR, the implicitly selected track is mostly not subject to editing or CAL actions unless it is also explicitly selected[38].

This selection characteristic can lead to problems when running a CAL program that operates on several tracks and must therefore be free to select and deselect these tracks as needed. As these tracks are selected, acted upon, deselected and other tracks selected, the implicit selected track will be equally affected each time. An example is a program that selects a range of tracks and sets all of the events on each track to a unique MIDI channel number. Because this program selects the tracks one at a time and then performs an interpolation on them, the implicit selected track is also interpolated over and over each time some other track is selected and operated on. The solution is to operate on all explicitly selected tracks first and then deselect all other tracks and interpolate the events in the implicit selected track last. This has the effect of **undoing** all of the undesired interpolations the track was subjected to while each of the others were selected and operated on. There may be cases when an action is carried out that cannot be **undone** in this manner. In cases such as this, it is important that the user ensures that the implicit selected track is an empty track so

[38] In SONAR the "**insert**" function inserts, if not in a "**forEachEvent**" loop, in the implicit selected track.

that as CAL selects tracks for editing, these explicitly selected tracks are the only ones with any data that will be affected.

5.6 Using the CAL View Window

The look and functionality of the CAL editor – for versions of Cakewalk before SONAR - has not changed much from version to version with the exception of one feature known as '**parentheses kissing**' which I will explain later and the addition in version 7 (finely!) of text cut and paste editing functions. Aside from this detail, the following discussion will apply equally for all versions of Cakewalk from 3 through 8.

5.6.1 Entering CAL Text

- The CAL editor has been optimized for writing CAL code in two ways. First, the tab key causes the cursor to indent just enough to help keep nested functions looking neat and readable. The indention is only two characters wide so it's possible to have long strings of multiply indented text and still not run off the end of the page. As you use the '**Enter**' key, the cursor will start on the next line at the same indention point as the line just above. This makes entering nested code a breeze. To back up one indention level, just hit the '**Backspace**' key once and the cursor will move to the left coming to rest at the previous indention point. This makes it easy to place the closing parentheses for functions at the proper indention points that correspond to their opening parentheses.

- The other optimization is the '**parentheses kissing**' feature. What this feature does is help the programmer check to see if the closing parenthesis they are entering corresponds to the expected opening parenthesis. The way this works is that as a closing parenthesis is typed in, the editor will briefly scroll up to and highlight the opening parenthesis that the editor associates with it. If the opening parenthesis is the one the programmer expected to see, then the code is '**in phase**' If the highlighted parenthesis is one that the programmer didn't expect to see, then the code is '**out of phase**' and the programmer has either left out or added in too many parentheses somewhere in the program. This is a very valuable check-up device, and it is a real shame that it was removed starting with version 5, and continuing with version 6. The problem is that the editor will only highlight an opening parenthesis if it is visible in the window along with the closing parenthesis being typed in. The editor will no longer scroll back through the code to display the section where the opening parenthesis is located if it's out of view. This flaw has been corrected for version 7.

- There is an array of buttons at the bottom of the CAL editor screen in versions 3 through 6. The '**Open**', '**Save**' and '**SaveAs**' buttons need no elaboration. They work much the same as for any Windows application. The '**New**' button clears the editor (you are prompted to save any unsaved work, of course) and displays a default template that consists of an opening "**(do**" function, an indented "NIL" place holder and a closing parenthesis. The "NIL" is highlighted blue, meaning that anything entering the editor either from the keyboard or by recording a macro will delete and replace it. Therefore, you can just start typing in your code and it will begin at the first indention point and on the first line directly beneath the "**(do**". So long as you do not force the cursor beyond it, your code will also stay just above the closing bracket for that opening "**(do**". The '**Run**' button starts execution of whatever code is present in the editor. The '**Record**' button is a bit different in that once it is set into motion, it changes into a '**Stop**' button. This button starts and stops the Macro recording feature of CAL.

- In versions 7 and 8, the CAL view has taken on the standard look of any Windows 95 text editor. All file and edit operations, including an '**Undo**' feature, are available from the standard drop-down menus at the top of the screen which take on a CAL context when the CAL view has focus (Its title bar is blue). From these menus, you can create a '**New**' work area, load and save files in the usual manner and access the text editing features. The '**Record**' '**Stop**' and

'**Run**' functions are now controlled by a set of transport buttons just above the text window and operate as one would expect.

- Just as you can now have multiple workspaces open in versions 7 and 8, you can have multiple CAL programs open too. You must be careful not only which CAL window has focus when you run a program, but also which song has focus. Otherwise, you could end up running the CAL program on the wrong song!

5.6.2 Hidden traps in the CAL editor

- From time to time, I will be writing code and get an error that I can't clear no matter what I do. For example, I may get a 'Missing one or more closing parentheses' error and not be able to find the offending part of the program. There were times when I have had to scrap the entire program and re-enter all of the code again in order to get it to run, even though the old and new code are identical to the eye. As it turns out, they may not be identical to the CAL interpreter. Things that the eye can't see in the editor window are plain as day to the interpreter.

  I have discovered that I could have what appears to be bullet-proof code and still not be able to clear a 'Missing one or more closing parentheses' error. In fact, I tried once to remove entire chunks of a program to make it go away only to be left with

```
(do

) ; end of do
```

  and still I got the error! I then added the "NIL" so that it read

```
(do

    NIL

) ; end of do
```

  just like you see when you first open the CAL view, and still it was there. I ended up saving the fragment to disk along with a copy of the same fragment after performing a '**New**', then looked at both with a hex editor to see why one would run (not do anything, but at least it would run) and the other wouldn't. What I discovered surprised the hell out of me! The only difference was that the working code had two line feeds at the end and mine did not. I went back to my full program, added two line feeds at the end and BEHOLD! It ran!

- Sometimes a 'CAL Error 001: Syntax error' will crop up and display a chunk of text from a comment line. I have found that using double semicolons at the start of the offending comment line will clear it. I do not know why, it just does. There have been other times when no matter what I do, I still got some sort of error or another, and the only way I could solve this problem was to remove some or all of the comment lines in the program. There have been times when I have had to wipe the screen clean and enter the code all over again! Again, I don't know why this is true, but it is. Perhaps a stray non-displaying byte somehow finds its way onto the screen. You cannot see it, but CAL does and simply cannot figure out what to do with it. The only solution is to clear some of the text (and presumably the bogus byte with it), and start over. This has not happened often, but it has happened more than once to me in both versions 3 and 6. It has also happened to other people I know.

- If you fail to indent the lines after the opening "**(do**" in a program by at least one tab space, you may get errors. It seems that CAL needs these tab characters in the code to keep the functions strait and thus interpret the code correctly. As a rule, this isn't a problem in that good programming practice will dictate the use of tabs to indent each nesting level as the code is entered and so this problem should never arise. However, if you get in a hurry to test some small fragment and in your haste fail to indent at least one tab space in every line except for the opening "**(do**" and closing "**)**", you may not get the code to run even if it is otherwise flawless.

- If you check the section where I discuss the "**do**" function, you will see mention of another strange quirk in the CAL system. I had a section of code in one program made up of cascading

"**if**" statements like so:

```
(if (something or another....)
   (if (some more stuff...)
      (if (yet another statement...)
         (do
             (a bunch of stuff...)
             (and so on...)
             (and so forth...)
          ) ; end of do
        ) ; end of if
    ) ; end of if
) ; end of if
```

For some reason, CAL would not see the final "**if**" or any of the code nested under it. I got no error message or any other indication of a problem, it was simply ignored regardless of the outcome of the "**if**" statement above it. In order to test for CAL even going to the code, I added a "**pause**" statement to stop the program and let me know if CAL had gotten that far. In order to use the "**pause**", I had to add a "**do**" to the stack of statements like so.

```
(if (something or another...)
  (if (some more stuff...)
    (do
       (pause "found the IF statement!")
       (if (yet another statement...)
           (do
                (a bunch of stuff...)
                (and so on...)
                (and so forth...)
           ); end of do
       ); end of if
    ); end of do
  ); end of if
); end of if
```

After adding the "**do**" and "**pause**", the program worked perfectly. I couldn't figure it out, but I went ahead and removed the extra "**do**" and "**pause**" as well as the closing "**)**" I had added for the "**do**" and again the program stopped working. I finely went back and put the "**do**" back in without the "**pause**" and everything was fine. I know that the "**do**" is unnecessary according to the syntax of CAL, and yet the program would not run without it. Go figure! If this ever happens to you while cascading a bunch of "**if**" functions, try adding a "**do**" to the spot where CAL seems to be getting lost. It might be all you need to make things right.

- Believe it or not, I've had situations where the order of variables in a comparison involving zero had an effect on the ability of a function to generate an outcome. This sometimes will not work:

```
(if (> 0 count)
```

but if you rewrite it like this:

```
(if (<= count 0)
```

it will run just fine. It seems that there are times when placing the zero first and then the other variable will cause trouble.

- I must mention one more thing. I started playing (a bit roughly, I must admit) with CAL in version 7 one day and caused it to enter some funky mode where every function caused a "CAL Error 001: Syntax error" to be reported. All I could do was to restart Cakewalk and everything was fine. I never could reproduce this problem a second time, but just to be safe, if you ever get into a situation where CAL or any other part of Cakewalk stops obeying the rules, save your work, shut down and restart. That does the trick 99% of the time. For those 1% cases, restart Windows!

- In case you might be curious, there are ways of crashing Cakewalk through the CAL editor. If, for example, you take a '**New**' CAL screen in version 6 and in place of the "NIL" type in (if) and hit the '**Run**' key, you will get a Windows system error and Cakewalk will shut down. If you really want to be malicious, try setting the variable "Now" to a very large 32 bit value! Supposedly, the marker variables are double words, which mean they should be able to hold any 32 bit number from 0 to over 4 million. However, if you try to set "Now" to any number near this maximum value, not only will Cakewalk generate a system error, but if you keep re-opening Cakewalk and doing it again, you'll eventually blow it right out of the water! When you restart Cakewalk the next time, you'll discover it has forgotten its MIDI input assignments. I gather Windows still thinks there is an open copy of Cakewalk running and refuses to open the same MIDI devices for another instance of the program. In any event, I thought it prudent to restart Windows when it happened to me. By the way, all of these problems were corrected in version 7.0. So far, I have not found anything in CAL that will make this baby fall.
- One other thing worth mentioning. Glen's fellow CAL hacker Dean Brewer in South Africa told him about a CAL program he had written to chase down duplicate events that wouldn't run even though it was, by all appearances, flawless. It kept giving him 'Expression too complex' errors. Glen did some playing around with it and discovered there is a limit to the number of functions a CAL program can handle. That limit is 256. If you have more than 256 functions, you will generate the error. Here is the layout of Deans program, edited for space:

```
(do
   (TrackSelect 1 -1)
   (= From 0)
   (= Thru 5)
   (a bunch of edit filter setup functions)
   (say, about 10 or 12 altogether)
   (EditDelete40 1 1 1 0 0 0 0)
   (TrackSelect 1 -1)
   (= From 0)
   (= Thru 5)
   (a bunch of edit filter setup functions)
   (say, about 10 or 12 altogether)
   (EditDelete40 1 1 1 0 0 0 0)
   (another section like those above)
   (more sections like those above)
   (more sections like those above)
   (more sections like those above)
```

```
      (more sections like those above)
   ) ;end of do
```

Altogether there were 11 such groups of functions, each group made up of anywhere between 8 and 15 functions. To make a long story short, there were 261 functions between the opening "**(do**" and the closing "**)**". I tried his suggested, though as yet untried fix of grouping each section in its own set of "**(do**" nests like so:

```
(do
   (do
      (TrackSelect 1 -1)
      (all of the functions in this group)
   ) ; end of do
   (do
      (another group of functions)
   ) ; end of do
   (do
      (another group)
   ) ; end of do
   (do
      (and the same for all 11 groups)
   ); end of do
) ; end of do
```

And so on and so on for all 11 groups. It ran without error. Now the program was made up of only 11 functions instead of 261 functions. It just so happened that each of those 11 functions nested between 8 and 15 other functions, but now CAL did not care. It was not a matter of how many functions total there were, but just how many nests there were. To further address this issue, I removed all of those "**(do**" nests and then counted 255 functions before adding the "**(do**". It still worked. I moved the "(do" one more function down to go between the 256th and 257th and the error returned. So long as only 256 functions were present, the 255 functions plus the one "**(do**" function nesting the rest of them, everything was cool and the program ran. There is likely a limit to the number of nested functions that a CAL program can deal with in this way also, thus putting an upper limit on the total number of functions of any configuration that CAL will tolerate, but I have not been able to find that limit. I do know that CAL version 3 will only allow 10 nesting levels (nests within nests within nests) and that Windows 32 bit versions will accept more than this before giving an error. Otherwise, the true limits of CAL are there for you to explore. Send me an email if you find another boundary and I will add the information to this document.

5.6.3 Recording Macros

Once the '**Record**' button in the CAL view[39] window is clicked, many of the mouse and keyboard actions you take involving the Menu toolbar or the '**Track'** View will result in the generation of CAL code in the editor workspace. As you have seen from the above discussion of '**Menu**' functions, there are many operations in CAL that correlate directly with actions that can be taken from the toolbar. When you click '**Record**', the CAL view window disappears and you are placed back in one of the sequencer views. From here, you can carry out normal operations on the

[39] Not available in SONAR anymore.

sequence oblivious to the fact that CAL is recording some of your actions as program code. This recording will continue until you switch back to the CAL view and click '**Stop**'.

What you will see in the editing window as a result of recording is first a comment line declaring '**Start of recording**' followed by any actions you took that are recordable as CAL functions. After these function statements is another comment line stating '**End of recording**'. All of the arguments for each function will be present in the form of numbers that reflect any data and whatever boxes were checked or in force at the time the menu command was invoked. As recorded, these CAL statements have very limited use outside the specific conditions at the time they were recorded. In order to use these macros to carry out their actions under different conditions, many of the arguments will have to be edited and replaced with variables. Once you're happy with whatever you have recorded and edited, you can save it as a '**.CAL**' file and can be run at will like an '**Edit**' menu feature.

5.6.3.1 Pre-setting Edit Functions Using Macro Record

As I mentioned earlier, setting up some of the '**Edit**' menu functions can be a real pain in the butt, especially if the filters are involved. We can save ourselves some major typing by using the Macro record feature to **pre-write** most of the needed code. What we would do is place the cursor at the point in the CAL program code where we want the function to go, and then click the '**Record**' button. After doing so, the display will leave the CAL view and go to the last view window you were in. Now select a track with some events in it and set the From and Thru markers to any part of the track containing events, if they aren't already. Go to the EDIT menu and do the following:

- For Cakewalk 3, click on the edit feature you want to put in your CAL program and set the check boxes and window values as you might want them. Check '**Use Event Filter**' if you will need to filter the edit. If you checked '**Use Event Filter**', you will be presented with the filter window. Make whatever selections you what CAL to use when your program is run. **Remember**, the settings you make here will be in force **every time** the program runs so make sure you know what you want the function to do! For any arguments that will eventually be filled in by variables in the program, don't worry what values they are set for now, as those values will be replaced during editing. Click '**Ok**' to complete the operation. After finishing with the edit, go to the '**Edit**' menu and click '**Undo**' to repair whatever you just did to your sequence (unless you like what you did and want to keep the change). Now flip back to the CAL view and click the '**Stop**' button.
- For Cakewalk 4 or higher, from the '**Edit**' menu, open the '**Select**' sub-menu and then the '**By Filter**' option. Set up your filter even if it is nothing more than clicking the '**All**' button. Now click on the editing feature you wish CAL to run and fill in the boxes and windows as needed. Otherwise, the same rules for the window contents apply as noted for Cakewalk version 3 as discussed above. Afterwards, perform the same '**Undo**' from the edit menu to restore your sequence and then stop the macro recording.
- Now you must edit the resulting functions CAL has placed in your program. Replace the numbers in the marker arguments with "From" and "Thru", the variable names you wish to use or the functions that will result in the times you wish to be in effect when the edit runs. Do the same for the values of 'min' and 'max' as needed for any '**SetFilterRange**' functions that require them. This also applies if you will be using variables for any of the other arguments.
- As we discussed in the chapter regarding '**Edit**' menu functions for Cakewalk version 6, a macro-created "**EditInterpolate**" function will only work for marker variables "From" equaling 0, and "Thru" equaling "End" regardless of what you change the "From" and "Thru" values to. In other words, the macro will operate on the entire length of the track and cannot be set to operate on only a segment instead, at least not directly. In order to force CAL to

recognize values for the markers other than 0 and "End", we must preface the "**EditInterpolate**" function with two **dummy** statements (see chapter 5.4: 'Forcing Version 6 (EditInterpolate) To Use Markers' for a discussion of this approach). Place these **dummy** statements just before each "**EditInterpolate**" function in your program. This will have no effect in versions other than 6, but will activate the necessary marker recognition in version 6 thus assuring portability of your code.

Note: that it appears that you must use these **dummy** statements for **every** such function, not just the first.

- Also remember that most CAL 4.0 functions need to have the markers set up beforehand and the CAL30 functions require the markers as arguments. Also remember that the CAL 3.0 functions will work in all versions of Cakewalk whereas CAL 4.0 functions will not work in Cakewalk version 3. If you want your code to be backward compatible and you record your macros in version 4 or above, you must rewrite the functions to turn them into CAL 3.0 format. Unfortunately, this may mean it will be easier to write the code by hand and forget about macros.

5.6.4 Ghost In The Machine

- If you ever enter a bunch of code into the editor and it simply will not run, don't be too alarmed. First of all, you must try to take the error message seriously and attempt to address the offending code. However, if you have followed all of the rules and the code still will not fly, you may have run into one of many strange effects of the CAL editor. See the chapter 5.6.2: 'Hidden traps in the CAL editor' for details. As far as legitimately issued error messages go, some of the more common ones are, 'Missing one or more closing parentheses', 'Expected closing quote', '**unknown procedure**', 'CAL Error 003: Wrong number of arguments', 'CAL Error 001: Syntax error' and the infamous 'Evaluation stack overflow'. If you really piss it off, you can end up with a Windows '**Protection Fault**' and/or Windows '**Illegal Operation error**', which will shut Cakewalk right down and maybe even Windows with it! This is not Cakewalk's fault. If you hose your code so badly that it chokes the system, then you probably deserved it! Cakewalk, properly installed, is very stable under Windows, and this is saying allot considering that Windows before Windows 2000 isn't exactly the Rock of Gibraltar. As far as the internal CAL errors, these can be dealt with.

- The error about the closing quotes is easy to fix. Just hunt through your code until you find a missing set of quote marks. There is nothing special about that. The "unknown procedure" and "syntax error" messages can usually be traced to a typo someplace. Perhaps you didn't use a capital letter when you needed to, or there may be a variable without a keyword someplace. Take your time looking for these things. If you're told you have the "wrong number of arguments", then look for an incomplete function. Another cause, believe it or not, can be the failure to indent properly. You should always indent at least one tab space away from the left margin after the opening "(do" no matter what. If you don't, this is one way CAL can express its dissatisfaction with you. If you import your text from a word processor, you might be dealing with hidden characters that will not display on the screen but never the less are seen by CAL at run time. If you can narrow the problem down to a line or two, it pays to just delete them and reenter the code.

- An 'Evaluation stack overflow' is a bit harder to clear. It could even mean scrapping the program all together! This message is an indication that CAL is being asked to juggle too many balls at once. It can result from a statement that is too complex. Splitting the statement up into a group of simpler nested functions can solve this. On the other hand, you may be declaring too many variables. Try to make some of them do double duty or use the "undef" function to clear some variables after they have fulfilled their destinies and then declare some more as needed. If you run too many nests in one function, this will cause it. You can tell if you have done so if you have reached a point where a statement is indented over so far as to be half way across the screen! If you can't reorganize your code to reduce the number of nests in this poor function,

you may have to rethink your entire project. Maybe you're asking too much of CAL and you need to create a simpler version of the program.

- This brings us to 'Missing one or more closing parentheses'. You might think this would be an easy error to clear, but it's not all the time. A missing parenthesis can be very hard to spot. Life is a bit easier using versions where "parentheses kissing" works. One can always backspace over a parenthesis and then re-enter it and see what opening parenthesis is highlighted. After going back through the code doing this every few lines, the out-of-phase statement will eventually show itself by causing the wrong opening parenthesis to highlight. Now, it's a bit more of a hassle with versions 5 and 6. This is a good reason to be very diligent in one's indenting. Sometimes the indention level being off is the big clue to where the missing parenthesis needs to go. What's more, there are times when this error can occur and there be nothing at all wrong with the parentheses. Take a look at the examples I cite in the chapter 5.6.2: 'Hidden traps in the CAL editor'.

5.7 Under the influence

Glen has several versions of Cakewalk on the same computer. Notably, he has version 3, version 6 and version 7. As a bit of background, he started out with version 3 on a 386 running Windows for Workgroups 3.11 and later installed it on a 486 running the same version of Windows. Over time, he installed Windows 95 release A on that 486. Around December of 1997, he obtained a copy of version 6.01 and installed it into another directory of the 486 and ran both versions so he could compare features as part of writing this CAL tutorial. He remembered noticing that the CAL editor of version 6 lacked the ability to perform '**parentheses**' when the opening parenthesis was off the screen. He was a bit disappointed with this because he found the feature very useful in debugging CAL code. This aside, he soon after upgraded to a Pentium and Windows 95 release B. Again, he installed both copies of Cakewalk on his system, first version 3 and then version 6. One day, he went back to version 3 to do use the fully operational parentheses kissing feature on a CAL program he was troubleshooting, and **it was gone**!!! That is to say that the feature in version 3 now works exactly like it does in version 6. Somehow, version 6 has **contaminated** version 3 to the point that version 3 has started taking on some of version 6's characteristics.

He did not know how this happened, but he tried it on another Pentium he had in the bedroom and sure enough, as soon as he installed version 6, version 3 lost parentheses kissing. He tried uninstalling version 6 and reinstalling version 3 but the contamination remains to this day on both computers. Even now that he had installed version 7 beta on one of them with parentheses kissing restored, he still has the problem with version 3. He has also tried installing version 3 on other other 386 and 486 computers running Windows 3.11 and everything is fine. He has installed on another 486 running Windows 95 release A and have full parentheses kissing. He even installed it on a Pentium Pro running Windows 95 release B and had no trouble. He then installed version 6 on the Pentium Pro system and version 3 was **completely unaffected**!! Go figure!

This may seem like a small thing to be worried about, but let me say that the parentheses kissing phenomenon is the only contamination effect he **has noticed**. Other subtle changes are maybe not noticed. I don't know how likely it is that this cross **contamination** would happen on your or any other system, but at least be aware that if you have multiple versions of Cakewalk on the same computer, odd things might happen.

5.8 CAL interactions with digital audio

This is going to be a small chapter in that I have only noticed one bit of odd behavior running a CAL program on a MIDI track and then having some strange effect on digital audio tracks. I have noticed in version 8 that allot of things will cause the audio effects '**Return**' knobs to reset to the full '**Off**' position if you fail to take an opening **snapshot** at the start of a project. Among these

causes is running CAL on a MIDI track with the Console View open even if that view does not have focus (its blue Title Bar is gray). It doesn't happen all of the time, just like closing a project and reopening it doesn't always move that knob to '**Off**', but sometimes it will. I can't lock down any repeatable pattern, it just happens once in a while. If you ever run a CAL program and after starting playback notice that your audio tracks have gone dry, perhaps the aux1 '**Return**' knob has been reset. I have made it a habit to record all audio Console View and effects settings in the song's '**Info**' page and keep it updated as I edit. It's good to also include any settings for your audio card mixer or outboard mixer too.

5.9 Details on RPN and NRPN Controller Events

If you want to keep the "RPN" event function and still want a convenient way of changing pitch bend depth from the Event List view, the following chart should prove valuable. To change the bend depth for most General MIDI compatible synthesizers, you would issue 3 controller events. The first would be controller number 100 with a value of 0. The second would be controller number 101 also with a value of 0, and the last would be controller number 6 with a value corresponding to the number of semitones you want the bend range to have. Some instruments also require a forth controller event to terminate the string. It is controller 38 with a value of 0. The "RPN" event that does the same thing will have two digits associated with it. The first will be a parameter value of 0 and the second will be a data value number equal to the number of semitones of bend depth multiplied by 128. For example, if you wanted to set the bend depth to plus or minus an octave, or 12 semitones, the controller sequence would be:

```
Controller 100, value of  0
Controller 101, value of  0
Controller 6, value of  12
Controller 38, value of  0 (not always needed)
```

The same job would be done by the single "RPN" event having a parameter value of 0 and a data value of 1536. The following chart outlines all RPN data values for bend depth ranges from +/- 1 to +/- 24 semitones. In all cases, the RPN parameter value is 0.

Bend Depth (+/- semitones)	RPN Data Valu(Depth X 128)
1	128
2	256
3	384
4	512
5	640
6	768
7	896
8	1024
9	1152
10	1280
11	1408

Bend Depth (+/- semitones)	RPN Data Valu(Depth X 128)
12	1536
13	1664
14	1792
15	1920
16	2048
17	2176
18	2304
19	2432
20	2560
21	2688
22	2816

Bend Depth (+/- semitones)	RPN Data Valu(Depth X 128)
23	2944

Bend Depth (+/- semitones)	RPN Data Valu(Depth X 128)
24	3072

If you have been using Cakewalk version 6 or higher, you may be aware of the "RPN" and "NRPN" events. These events are clusters of combined multiple "CONTROL" events that have been converted into a single event.

The drawback is that most of us are used to working with the controller events **as is** and have no desire to learn some obnoxious formula to convert "RPN" events back and forth when we need to view or edit them. An example is the way Cakewalk will take the three controller events that make up a pitch bend depth setting message and stuff them into one "RPN" event.

What if you want to change the bend depth on that track? How do you change this "RPN" event to reflect the change in the value byte of the third controller event? Well, without a calculator and a few minutes of valuable time, you cannot. But more important is that CAL programs cannot access these events anymore. However, there is a way around the problem.

By adding a statement to the '**TTSSEQ.INI**' file in the CAKEWALK\SONAR directory, we can instruct Cakewalk not to combine clusters of controller events into "RPN" and "NRPN" events. Open '**TTSSEQ.INI**' in the **Notepad** editor and locate the heading [OPTIONS]. Under this heading add the line,

```
TranslateRPN=0
```

and save the file. Now restart Cakewalk and the "RPN" event will never darken your sequence again! Note that if your sequence has native "RPN" events already in it, they will appear as such. This change does not force Cakewalk to decompile "RPN" events that are present in a sequence, just prevents it from generating them out of clusters of related controller events. For those who want to keep the "RPN" event and still want to change bend depth from the Event View, see above for a chart of "RPN" values that correlate to pitch bend depth values.

5.10 Helpful Tips for Using the DLL Function

By Mehmet Okonsar Aug 3, 1999

General : The experiments were done with Cakewalk ver.8 and Delphi3.

5.10.1 Data Types:

INTEGERS : Data sent by CAL as type "**integer**" must be declared as '**byte**', '**word**', '**smallint**', '**longint**', etc in the "**DLL**" code. Random data is received if the "**DLL**" declares integer too.

STRING : Any strings must be declared as null terminated '**char**' arrays in the "**DLL**".

5.10.2 Parameters:

One can send 2 parameters at most when calling a service in a "**DLL**". The third parameter, if any, does not generate an error but random data is sent.

5.10.3 Memory Management:

As far as I know there is no way to make the call to "**DLL**" **persistent** or **static**, even though the "**DLL**" is called by SONAR (not the CAL program), the "**DLL**" is removed from memory as soon as the corresponding line in the CAL code has been processed. This is because CAL is interpreted and not compiled, and it does not provide the possibility to make a persistent call to a "**DLL**". This fact, combined with CAL's own limitations, makes it impossible to send a bunch of data to the "**DLL**" all at once. If the "**DLL**" is to be used to write to a file, SONAR keeps **hands** on the file which can be opened, written to and closed only once and then cannot be reopened again without closing SONAR altogether. There is no problem calling the "**DLL**" from within a loop, though this will slow down the program.

5.10.4 Miscellaneous:

The arguments for the call to a "**DLL**" must include the full path name even if the CAL file making the call is in the same directory with the "**DLL**". This path can be either a long name or an '**8.3 DOS**' type path names. It seems, they both work. For **safety reasons** I kept with the DOS type definitions.

6 CAL Error and FYI messages

Before CAL even starts to run a program, it goes through and makes sure the code is correct enough to convert into machine instructions. If there are errors that prevent the program from being interpreted, CAL stops there and issues a message. Until now, nothing has been done to the sequence. On the other hand, if the code can be at least interpreted, then CAL starts running it. If there is an error at a certain point in the program, CAL may very well have already done something to your sequence and so an '**Undo**' will be necessary to return the status quo before you attempt to fix and run the program again. To be safe, if a CAL program doesn't perform exactly as expected, perform as many '**Undo**' operations as necessary to erase all actions taken by CAL.

Another approach is to save your sequence, and then run the CAL program. If there is an error use the '**Escape**' key and reload your sequence without saving the **damaged** one.

Glen has taken this list of error messages from examining the text portions of Cakewalk's version 3 program file WINCAKE.EXE with MS Word and hunting for the areas that contain CAL related text. It may not be complete, may contain errors that don't belong to CAL and may not reflect the full range of error messages available to later versions of Cakewalk. However, with no documentation (as is true with a great deal of CAL) and my own experience torturing CAL, this is what he and I have come up with.

Attempt to change constant

Naturally, you cannot re-assign the value of a constant like "NOTE" or a read-only variable "TIMEBASE", and attempting to do so leads to this error. As a rule, constants in CAL are in all caps to make them easy to distinguish from variables.

CAL Error 001: Syntax error

CAL understands your words, but not the way you have them arranged. Perhaps you have confused a keyword with a variable or arranged the components of a statement incorrectly.

CAL Error 002: Divide by zero

Needless to say, if you ask CAL to divide by zero, it won't be happy. Rather than give your processor a coronary trying, CAL traps such equations and issues this error message.

CAL Error 003: Wrong number of arguments *function_name*[40]

Unless you genuinely left out an argument in a statement, this error is usually caused by not having the closing parenthesis of a nested section of code in the proper spot. You likely have the parenthesis somewhere; otherwise you would receive a "Missing one or more closing parentheses" error. Maybe you just put it in the wrong place and so now some "**if**" function has a '**then**', an '**else**' and a stray extra part, or a mathematical statement is being asked to operate on three arguments instead of two.

[40] If the error is within a menu function you will not get the funcion_name.

CAL Error 004:Unknown procedure *procedure_name*

Several things can cause this. You may have mistyped a keyword. Perhaps you have two statement elements running together without a space between them like (+this that) instead of (+ this that). Misarranged parentheses or sets of parentheses without a space between them will also cause this error. Basically, CAL cannot understand some word in your code.

CAL Error 010: Types do not match

If you try to use variables for functions unsuited for their type class, CAL will let you know. If you attempt to place a negative number in a "word" variable or try to perform time functions on an integer variable, you are likely to see this error.

CAL Error 014: Value out of range

This error message lets you know that you have attempted to generate a value that exceeds the range of the variable. Keep in mind that even though an integer can hold a certain value, the variables relating to your sequence may not. E.g. a "Note.Key" must be between 0 and 127.

CAL Error 021 Program called (exit)

This message is not an error, but confirms that CAL has executed an "**exit**" function and is aborting.

CAL Error 022: User pressed cancel

This is not an error either, but confirms that CAL has intercepted the user pressing the '**Escape**' key or clicking a '**Cancel**' box from a dialog box and is therefore shutting down.

CAL Error 023: Cannot open include file *file_name*

CAL cannot find the file you have asked to be included in a program. This is likely because of a mistyped file name or because the file in question is in a directory other than the one where CAL is looking. CAL will look in the directory where the currently running program is stored if no path is specified. If a full path is given, CAL will look there instead.

Cannot load Dynamic Link Library

You have asked CAL to use a service from a "**DLL**" file, but CAL cannot find the file. The causes are the same as listed above.

Command is disabled on the menu

Here, you have asked CAL to perform a '**Menu**' function that is currently greyed out in the menu's drop-down list such as attempting to do an '**EditPaste**' with nothing in the clipboard.

Evaluation stack overflow

There is a limit on not only how complex a line of code can be, but also on how many nest levels you can build in one function. If you keep nesting "**if**" after "**if**" after "**do**" after "**do**", sooner or later you will get this error. Having too many variables declared, will also generate this response.

Expression too complex

It is one thing to try to be efficient by putting as much code into one function as possible. It is something else entirely to build expressions that are so **compound** that CAL cannot run them. Do not try to do everything in one line of code. Use the nest maker "**do**" if you need to do much in one argument or having a large number of functions at the same level.

Expected closing quote

This is plane enough. If you use quotes to enclose a string or a message, then forgetting one or more of them will cause this error.

Miscellaneous error

I've never seen this, but I guess if you stump CAL completely, you may receive it.

Mismatched parentheses

Assuming you have the correct number of opening and closing parentheses, you may not have arranged them properly. This error message tells you to re-examine your code for out-of-place parentheses.

Missing one or more closing parentheses

This is the message you are likely to see most often. CAL counts the number of left brackets and right brackets in your program, and if the numbers do not match, you get this error. Not getting this error does not guarantee that your parentheses are on the right places, just that the counts match. You can also get this error by not having 2 line feeds following the last closing parenthesis at the end of your program.

Not valid in (forEachEvent) or body expression

This is the opposite of the above error. There are some things that you just can't do while CAL is scanning a sequence. An example is attempting to run '**Edit**' menu functions during a loop.

Out of memory

I have never seen this error, and I have written some **big** programs. I guess a point can be reached where you overflow the space CAL has to work with or insert so many new notes that your sequence is too big to stay in memory. Go buy some more **ram**. It's just too cheap these days to be caught with not enough memory to get a job done.

Proc does not exist in Dynamic Link Library

CAL found the "**DLL**" file all right, but you have supplied arguments that do not point to a valid service within that file.

Program called (error)

If you set up a function to call the "**error**" function in the event of some condition, CAL will display this message and then abort.

Undef of undefined variable

If you attempt to execute the "**undef**" function to erase a variable from memory and there is no such variable to begin with, you will see this error.

Unknown variable

Apparently you either attempted to use a variable name that hasn't been declared or you mistyped the name of one that has.

Valid only in (forEachEvent) or body expression

The operation you are trying to perform can only be done within a "**forEachEvent**" loop. These are things like the "**delete**" function.

Variable redefined

You have attempted to declare a variable that has already been declared. One way to get this error is if you include a program that declares a variable with the same name as one declared in the parent program. To keep this from happening, make it a habit to use different capitalization for variables you declare in programs you intend to "**include**" in other programs.

7 Examples

The example programs can be found in the zip-file 'CAL program samples.zip'.

Addition.cal	
Purpose	To show the basic syntax of CAL with user interactions.
Usage	Adds two integers entered by the user.
Remark	None
MIDI-functions	None

GM Mode.cal	
Purpose	Switches the GM mode of the synthesizer.
Usage	Run the program: Enter the GM Mode: GM1=1, GM2=2, GM off=0.
Remark	
MIDI-functions	System exclusive message

Program skeleton.cal	
Purpose	The base for creating a program.
Usage	Open the skeleton, and save it with the name of the to be created program.
Remark	Adapt the skeleton to your own needs. Rename it, because otherwise it may be replace by my skeleton, when you get a new version of the sample file.
MIDI-functions	None

Show version.cal	
Purpose	To show the structure of programs with program header and comments.
Usage	Running the program will display the version of CAL.
Remark	None
MIDI-functions	None

Timebase.cal	
Purpose	To show the "TIMEBASE" value of CAL.
Usage	Running the program will display the "TIMEBASE" value.

Timebase.cal	
Remark	In certain (sub) versions of SONAR "TIMEBASE" does not match with the '**ticks per quarter note**', which can be set by the user. This program allows you to check whether "TIMEBASE" is the same as the setting of '**ticks per quarter note**'.
MIDI-functions	None

8 Library

The library programs can be found in the zip-file 'MIDI-Kit library.zip'.

+2-2 meter.cal	
Purpose	Indicates a change in the velocity of marked notes according to the position in a measure with $^{2}/_{2}$ meter.
Input parameters	Integer '**nVelRef**': Velocity reference (0 … 127) Integer '**nDeltaVel**': Random delta on velocity in % (0 … 5)
Output parameters	None, however, the note velocity of marked notes have been changed.
Remark	Note positions must be within grid.
MIDI-functions	None.

+3-4 meter.cal	
Purpose	Indicates a change in the velocity of marked notes according to the position in a measure with ¾ meter.
Input parameters	Integer '**nVelRef**': Velocity reference (0 … 127) Integer '**nDeltaVel**': Random delta on velocity in % (0 … 5)
Output parameters	None, however, the note velocity of marked notes have been changed.
Remark	Note positions must be within grid.
MIDI-functions	None.

+4-4 meter.cal	
Purpose	Indicates a change in the velocity of marked notes according to the position in a measure with $^{4}/_{4}$ meter.
Input parameters	Integer '**nVelRef**': Velocity reference (0 … 127) Integer '**nDeltaVel**': Random delta on velocity in % (0 … 5)
Output parameters	None, however, the note velocity of marked notes have been changed.
Remark	Note positions must be within grid.
MIDI-functions	None.

+6-8 meter.cal	
Purpose	Indicates a change in the velocity of marked notes according to the position in a measure with $^{6}/_{8}$ meter.
Input parameters	Integer '**nVelRef**': Velocity reference (0 … 127) Integer '**nDeltaVel**': Random delta on velocity in % (0 … 5)
Output parameters	None, however, the note velocity of marked notes have been changed.
Remark	Note positions must be within grid.
MIDI-functions	None.

+9-8 meter.cal	
Purpose	Indicates a change in the velocity of marked notes according to the position in a measure with $^{9}/_{8}$ meter.
Input parameters	Integer '**nVelRef**': Velocity reference (0 … 127) Integer '**nDeltaVel**': Random delta on velocity in % (0 … 5)
Output parameters	None, however, the note velocity of marked notes have been changed.
Remark	Note positions must be within grid.
MIDI-functions	None.

+Constant.cal	
Purpose	To provide project constants which are not standard included in CAL.
Input parameters	None.
Output parameters	Word '**BEATTICKS**': Beat duration in ticks. Integer '**MEASBEATS**': Measure duration in beats. Word '**MEASTICKS**': Measure duration in ticks.
Remark	The best place for this include program is before the declarations of the variables in the calling program. Only one meter in the sequence is supported; it will use the meter at the start of the sequence. **Note 1:** The provided constants are not really constants, but variables. You must not change the value within your program. **Note 2: Not for new developments**. The "+Meter.cal" include program must be used. It provides more functionality.

+Constant.cal	
MIDI-functions	None.

+Controller.cal	
Purpose	To insert or replace MIDI controllers at the marked area of one track..
Input parameters	Integer '**nControlNum**': Controller number String '**tControlName**': Controller name
Output parameters	Integer '**nDirection**': indicates whether controlers went up (1), down (-1) or did not change (0)
Remark	The user of the calling program must mark the area in a track where to insert the controllers. **Option 1:** Mark the note(s) and/or event(s) in the staff or event list panel where the controller(s) must be inserted: Run 'the program', and Enter the Controller number[41] (0 .. 127); Enter the choice for single insert or multiple inserts (0..1); Enter the value of the controller (0.. 127); For multiple inserts enter the end value of the controller; For multiple inserts enter the beat value (1 ..64). **Option 2:** Mark the range where you want to insert the controllers: Run 'the program', Enter the Controller number (1..127), Enter the requested start value (0..127), End value (0..127), and beat value (1..64). The beat value indicates how the controllers will be inserted. The program calculates the insert time by dividing the beat with the entered value.
MIDI-functions	Dependent on input.

+Events marked.cal	
Purpose	To count number of marked events.
Input parameters	Integer '**nEvent**': Event kind
Output parameters	Integer '**nEventCount**': Number of Events
Remark	Events with the same "Event.Time" in the same track are counted as **one**

[41] Can also be provided by the calling program itself.

+Events marked.cal	
	event.
MIDI-functions	None.

+Meter.cal	
Purpose	To provide meter variables which are not standard included in CAL. The content of the variables are dependent on the position in the MIDI sequence.
Input parameters	Double word '**rPosition**': Position in the sequence in raw time
Output parameters	Word '**BeatTicks**': Beat duration in ticks. Integer '**MeasBeats**': Measure duration in beats. Word '**MeasTicks**': Measure duration in ticks. Word '**Meter**': meter in the format 100*X+Y. In which X/Y indicates the meter. E.g. ¾ becomes 304.
Remark	This include program requires, that the variables are declared with the include program "+Meter declarations.cal". This include program can be used multiple times in a(n include) program. Multiple meters within a MIDI sequence are supported.
MIDI-functions	None.

+Meter declarations.cal	
Purpose	To declare the public variables for the include program "+Meter.cal".
Input parameters	None.
Output parameters	Word '**BeatTicks**': Beat duration in ticks. Integer '**MeasBeats**': Measure duration in beats. Word '**MeasTicks**': Measure duration in ticks. Word '**Meter**': meter in the format 100*X+Y. In which X/Y indicates the meter. E.g. ¾ becomes 304.
Remark	The best place for this include program is before the declarations of the variables in the calling program. Only one meter in the sequence is supported; it will use the meter at the start of the sequence. **Note:** These provided variables will me filled in by the include program "+Meter.cal".
MIDI-functions	None.

+need version.cal	
Purpose	To test whether the version number is equal or higher than the indicated number.
Input parameters	Integer '**nCALVersion**': the version number – without the dot - on which must be tested.
Output parameters	None.
Remark	If the version of CAL is too low, the program gives a message to the user, and terminates.
MIDI-functions	None.

9 Information sources

- Cakewalk Pro Audio User's Guide;
- Cakewalk SONAR User's Guide;
- Cakewalk Pro Audio 'Help' files;
- Cakewalk SONAR 'Help' Files;
- Cakewalk Power!; Scott R. Garrigus;
- Windows Multi Media Programming Reference.

10 Document conventions

A consistent method of highlighting is used in this document. Here, I give the approach:

'Normal between single quotes'	Groups words, to mark it as a single item;
Bold	Identifies important items;
'**Bold between single quotes**'	Refers to SONAR functions, SONAR error messages, parameter names, CAL files, Windows functions, Windows error messages;
"**Blue Bold between double quotes**"	Refers to CAL functions;
Underlined items	Identifies hyperlinks.

10.1 Backus Nauer Form

The notation of the functions of CAL are partly following the the Backus Nauer Form (BNF). Here we will give the used symbols of BNF.

<X>	Identifies an operand;
[X]	X is optional. Optional operands are denoted by [];
[X [Y]]	X and Y are optional. The sequence is important.
[X [Y] …]	X and Y are optional. The sequence is important. More parameters are possible.
<X>…<Y>	X and Y are required. The number of operands depends on the function.

11 Index