DIAL
DIagrammatic Animation Language
Tutorial and Reference Manual

Steven Feiner

Department of Computer Science
450 Computer Science Building
Columbia University
New York, NY 10027

feiner@cs.columbia.edu

Technical Report CUCS–276-87
September 1987
(Revised February 1989)

Abstract

This is a tutorial and reference manual for DIAL, a diagrammatic animation language that allows parallel processes to be represented in a compact graphical "time line". DIAL provides a soft animation machine whose instruction set is determined by a user-provided backend. Unlike more recent animation notations that rely on bitmapped displays, and hence on special editors and extensive run-time support, DIAL requires only a conventional full-screen editor for editing and no run-time support beyond its compact interpreter. DIAL is implemented in C and runs on a wide variety of System V and Berkeley versions of UNIX.

Keywords: computer graphics, animation languages, visual languages.

DIAL
DIagrammatic Animation Language
Tutorial and Reference Manual†

Steven Feiner

Department of Computer Science
Columbia University
New York, NY 10027

feiner@cs.columbia.edu

1. Introduction

This is a tutorial and reference manual for DIAL, a diagrammatic animation language that allows parallel processes to be represented in a compact graphical "time line". The 2D notation employed may be edited and displayed on any ASCII terminal. DIAL was originally developed to serve as a convenient testbed for research on realtime animation effects for raster graphics systems, and was used to animate pages of pictures and text in the IGD hypermedia system described in [FEIN82a]. It has since been used for animating higher-dimensional mathematical surfaces [FEIN82b] and as a general purpose tool for scripted animation.

DIAL's frontend interpreter is implemented separately from the backend packages of animation capabilities that it supports. All graphics operations are performed by a backend. The frontend only controls when the backend's routines will be called, based on the contents of an animation script. Thus, individual programmers may develop packages of animation instructions that implement the particular animation capabilities desired. In fact, there is nothing in the DIAL interpreter that restricts its use to computer graphics animation. It may be used equally well for any application that requires scripted control of a set of parallel processes, such as music synthesis or lighting control.

The remainder of this document is divided into several sections, the first of which is a tutorial for prospective users. The next section is a reference manual for the scripting language. The last section is a reference manual for the frontend interface and is intended for programmers with UNIX and C experience who want to create or modify an instruction set backend.

2. DIAL Tutorial

2.1. Overview

In DIAL, animation is accomplished through a set of *instructions* for a virtual "animation machine," defined by a particular backend. A few sample instructions from

†The original implementation of DIAL was accomplished while the author was at Brown University and was supported in part by the Office of Naval Research under Contract No. N00014-78-C-0396, a grant from The Foxboro Company, and an IBM Graduate Fellowship.

one backend will be discussed as DIAL's syntax is explained. Note that these instructions are not part of DIAL itself, but are part of a particular backend.

Each instruction in a typical backend may be passed parameters that specify how to modify the displayed image and its underlying data structure representation. For example, a **moverel** instruction may provide the capability to move a named object to a new location specified as an offset relative to its current position. An *event* is an instruction whose actual parameters have been specified – for example, a **moverel** instruction for a specific object and offset. DIAL provides the ability to define events in terms of the instructions supported by the backend and to indicate when and for how long they are to execute.

In DIAL, time is divided into small, finite-length *ticks*. Each event starts at a particular tick and lasts for some integral number of ticks. The granularity implied by these finite length ticks is realized by having the backend update the display (and relevant data structures) at the end of each tick, leaving a visual record of the event's incremental execution. For example, a **moverel** event that lasts for one tick would result in an object being moved to the specified new location at the end of the tick. A **moverel** event that lasts for several ticks would result in the object being moved incrementally to successive locations between its original location and the specified destination.

DIAL handles all flow of control by determining which events are to be executed during each tick. Backend routines are called with the length of the event, the current tick in the event's execution, and the event's parameters. These routines compute and execute the intermediate alterations to the display and data structure by interpolating between current and desired values during the execution of the event. Depending upon the event being executed, the interpolation may occur between spatial coordinates, the values associated with color table indices, etc. All interpolation calculations are performed by the backend, based on the information passed to it by DIAL's frontend.

2.2. Defining and Executing Events

The horizontal dimension of DIAL's 2D language represents time, with each column corresponding to a tick. Entries in each column of certain lines specify the events that are to be performed during that column's tick. Consider, for example, the simple animation script of Figure 1. The first three lines (including the blank line) are *comment lines*. Comment lines begin with a space, tab, or newline in column one (i.e., the line's first column). They may occur anywhere in the script, may contain any characters, and

```
        Move an object the same distance 6 times, each time more
        slowly than the last.

% throw moverel "ball" 0.0 0.1

throw   | |-  |--   |---    |----     |-----
```

Figure 1. A simple DIAL script.

are ignored by the system.

The fourth line is a *definition line* for an event, named **throw**, that will perform a relative move instruction, **moverel**, when the event is executed. Definition lines begin with a "%" in column one, followed by an event name, an instruction name, and any parameters. Note that this line defines the event, but does not cause it to be executed.

The last line in the example is an *execution line* that specifies a sequence of successively executed events. Execution lines begin with an event name. Each column in an execution line (starting at column nine, the first tab stop) corresponds to a tick during which an event might be executed. In the example, the appearance of the *execution character* "|" in the first tick's column causes **throw** to be executed during the first tick. The blank in the second tick's column indicates that nothing is to be done during that tick. The first **throw** event in the above example starts and finishes execution during the first tick and is one tick long.

The "|" in the third tick's column is followed by a "-". This is a *continuation character*. The appearance of a continuation character after the execution character means that the event's execution is to be continued or extended into the next tick. The second occurrence of the **throw** event above lasts two ticks; the third, three; and so on, for a total of six events. As well, the number of "blank" ticks between events increases with each successive event in the example, so that the time between the termination of the previous event and the onset of the next also increases.

In DIAL no numbers are needed to indicate the number of ticks during or between events. Much as the analog display of a clock with hands is often preferable to a numeric digital readout, the visually obvious relative durations expressed in this notation can be more telling than numeric procedure parameters. When exact starting and stopping ticks and event durations are needed, however, execution lines may be placed below comments that serve as ruled scales.

2.3. Expressing Parallelism

All of the events in Figure 1 are executed sequentially. Figure 2 shows how parallelism is notated. This example contains a number of definitions for **mix** events that change the values in color table locations. The **mix** instruction, which was developed as part of one of DIAL's first backends, takes three parameters: a destination color table index, the number of sequential destination entries to affect, and a source color table index. When a **mix** event has completed, its destination colors will have been replaced with the source color. If the event lasts longer than one tick, the replacement is done gradually by mixing the destination values with successively greater amounts of the source value, interpolating the colors between their original and final values. The negative color table values refer to special "read-only" colors that are not affected by color table manipulation. Optional parameters to the **mix** instruction allow blending only a percentage of the source color in the final mix and matching multiple source colors with destination colors.

Any group of contiguous execution lines, optionally separated only by nonblank comments, will be executed in parallel. The first four execution lines of Figure 2 form such a group. Two-thirds of the way into the execution of the **fade51** event, **swap52** starts execution. The **reset51** and **fade52** events likewise overlap their preceding

```
        One color is faded to background while a second is faded
        to another color.  The first is then faded back to its
        original color while the second is faded to background.

% fade51  mix 51  1  -1
% swap52  mix 52  1  -2
% reset51 mix 51  1 -51
% fade52  mix 52  1  -1
% fadeall mix  1 64  -1

                 1         2         3         4         5         6
        12345678901234567890123456789012345678901234567890123456789
fade51  |-----------------
reset51                          |-------------------
swap52              |-----------------
fade52                                        |-------------------

        Now fade the entire picture to the background color.

                 7         8         9        10        11        12
        12345678901234567890123456789012345678901234567890123456789
fadeall       |-----------------------------------------------
```

Figure 2. A DIAL script with parallel execution lines.

events. Here subtle timing differences may be easily determined by eye.

Imagine the entire script to be represented by a single set of parallel horizontal execution lines. Since this notation would be awkward to view or edit, DIAL allows the set of lines to be "folded" to fit on a page or display of finite width by slicing them vertically into sections placed one after the other vertically down the page. We call each of these sections a *staff* (pl. *staves*), after the set of horizontal lines used in conventional western musical notation. The execution lines in a staff may execute any previously defined events, may occur in any order, and may be optionally separated by nonblank comments. An event's execution line may not be duplicated in a staff. The first appearance of a definition or a blank line will begin a new staff. Since nonblank comments may be interspersed throughout a staff, animators may temporarily disable execution lines by indenting them and annotate interesting events with inline comments.

DIAL simulates parallelism by incrementally executing events in the order in which their execution lines appear in the staff. Thus, since execution order is well defined, noncommutative operations such as parallel rotations may be specified with predictable results. If desired, the relative order of execution lines may change between two staves. There may be as many staves as needed in a script. Figure 2 has two staves, the second of which has just a single line for the **`fadeall`** event that fades all of the colors to background.

2.4. Advanced Features

Figure 3 demonstrates several additional facilities provided by DIAL. Here, unlike previous examples, the execution line for **`M.bus`** contains blanks interspersed with its

```
% M.bus   moverel "flxible" 0.5   0.0
% M.car   moverel "porsche" 1.0   0.0
% M.fly   moverel "medfly"  0.02 -0.05

% Fade mix 1 64 -1


        Move the bus and car along the road, while fading all colors.
        Move the fly every now and then.

M.bus   |-----------            ------------            ------------
M.car   |-------------------------------------------------------------
M.fly            | ||  ||--- |---     |-----
Fade    |-------------------------------------------------------------

! time "on"

M.bus   |--------------      ----------
Fade    ----------------------------------------
M.fly            | ||  ||--- |---     |-----

! time "off"

        Move the fly very slowly diagonally upwards as it grows.

% M.fly   moverel "medfly" 0.2 0.2;  scale "medfly" 1.5
# 10

M.fly   #==========---------
Fade                      ---------------------- --- -- -
```

Figure 3. Advanced features.

continuation characters. During those ticks for which blanks appear, the event is "suspended" and will resume execution only during those ticks for which a continuation character appears. An event starts at the occurrence of an execution character in an execution line and terminates at the last continuation character before its next execution character (if any). Thus, the number of ticks for which the event executes is the number of continuation characters plus one for the execution character itself. Events may span staff boundaries, making possible events, such as **Fade**, that continue over several staves, much like a musical "pedal point."

The *immediate line* is a convenient variant of the definition line that begins with an "!". It has no name and is immediately executed "in place" as a one-tick-long event. This facility allows events that are typically executed only once, such as modal toggles, to be defined and executed on the same line. For example, the **time** instruction used above causes timing information to be output at the end of each tick.

An event may be associated with multiple instructions, allowing several events that would normally occur simultaneously to be referenced by a single name. For example, the redefinition of **M.fly** specifies two instructions, both of which will be executed during each tick of the event in the order in which they appear. Events are separated by

"`;`"s and may be continued onto any number of following lines as long as all but the last end with a "`;`".

Specifying long event executions can be quite tedious. Therefore, a shorthand notation has been provided, using the special characters, "`#`" and "`=`", shown in the final staff. These are, respectively, DIAL's *super execution* and *super continuation* characters, which allow a column to represent more than one tick. The number of ticks represented by these "super ticks" is specified by the most recent line beginning with the super execution character, "`#`", and followed by the desired number of ticks. The appearance of such a line setting the length of super characters ends a staff. In Figure 2 the length of a super tick is set to 10 in the line following the last definition of **`M.fly`**. Conceptually, each "`#`" is expanded to "`|---------`" and each "`=`" is expanded to "`----------`". If a super execution or continuation character appears in a column with a regular execution or continuation character, the regular characters act as if they were padded on the right with blanks.

Although there is no "super blank" character, a user can get a similar effect by including in each staff an execution line for a **`no-op`** instruction (which must be provided by the backend) and placing super characters in the desired columns. For example, if the **`no-op`** lines contained a single super execution character in the first execution column of an execution line and a super continuation character in every column thereafter, all columns of the script would be "super columns." Long animation sequences may be quickly previewed by using super ticks whose length is temporarily set to one.

Note that increasing the length of a supertick only increases the number of ticks executed during an animation, not the length of time represented by a single tick. Typical backends provide instructions that let the user determine the length of a single tick, e.g., by sending timing delays to the graphics system.

3. DIAL User's Reference Manual

3.1. Lexical conventions

Tab characters may appear wherever blanks are allowed. Each is interpreted as representing the appropriate number of spaces, assuming tab stops every eight columns. Therefore, the word "blank" will be used here to refer to either a tab or space. All tokens may be separated by an arbitrary number of blanks, except in *execution lines*, in which precise column positions are significant. There are no reserved words.

3.2. Scripts

A DIAL *script* is a file containing *comment lines*, *definition lines*, *immediate lines*, *definition continuation lines*, *super tick set lines*, and *execution lines*. Execution lines are grouped together in *staves* to support parallelism.

3.3. Comment lines

A *comment line* is a line whose first character is a blank or whose only character is a newline. Comment lines may contain any other character. They are commonly used not only to document a script, but to temporarily disable a definition line, immediate line, super tick set line, or execution line by indenting it or blanking out its first character.

3.4. Definition lines

A *definition line* begins with a "%" in column one, followed by an *event name* and any number of semicolon-separated *event definitions*. A definition line names and defines one or more events, but does not cause them to be executed. An event name may be reused as many times as desired in a script. A definition's scope lasts until the next definition line with the same event name.

3.4.1. Event names

An *event name* may be up to 7 characters in length. It may contain any character other than space and tab, but may not begin with a "#", "%", or "!".

3.4.2. Event definitions

An *event definition* consists of an *instruction name*, optionally followed by *parameters*.

3.4.3. Instruction names

An *instruction name* is the arbitrarily long name of one of the instructions provided by the backend. It may contain any character other than space, tab, or ";".

3.4.4. Parameters

DIAL allows three kinds of *parameter*: integer, floating point, and string.

3.4.4.1. Integer and floating point parameters

Integer and floating point parameters should be in the same form accepted by C and are stored as C ints and floats, respectively. Integers may be either decimal or hexadecimal.

Hexadecimal numbers must be prefixed by "`0x`" or "`0X`". Since parsing is done with the C library procedures *scanf()*, *atoi()*, and *atof()* in the current implementation, no error checking is done. Hence, a number is terminated by the first invalid character. For example, if an integer is expected then "`1f`" (without a preceding "`0x`" or "`0X`") is treated as the decimal integer "`1`", and not the hexadecimal integer "`1f`".

3.4.4.2. String parameters

Strings must be enclosed in double quotes ("). To get a double quote inside a string it must be preceded by a backslash (\). Any character may be preceded by a backslash, except for the newline character, which may not be used in a string. These backslashes will remain in the string, however. The backend must copy and convert the string if the equivalent of C's backslash escapes is desired. Therefore, a DIAL string parameter is exactly like a C string, except for the prohibition against escaped newlines and the requirement that the backend remove and interpret embedded backslashes if desired.

3.4.5. Multiple event definitions

More than one event definition may be placed on an event definition line, causing a single execution line to execute several instructions. If the first event definition is terminated with a "`;`", it may be followed by additional ones, each of which should consist of an instruction name, optional parameters and a "`;`" if another event definition is to follow. Blanks are optional on either side of the "`;`". The events will be executed in the order in which they appear on the line. The first line of Figure 4 shows **`event1`** being defined so that it will execute three instructions.

3.5. Immediate lines

An *immediate line* looks like a definition line, except that a "`!`" appears instead of a "`%`" and an event name is not used. An immediate line defines one or more unnamed events that are executed immediately for a single tick. Immediate lines provide a convenient way to specify events that need be executed only once.

Animators often include in their scripts a "menu" of comment lines, each of which may be selectively turned into an immediate event by placing a "`!`" in its first column. These are commonly used to specify modal settings in the animation system. Should it be necessary for timing purposes to not count such a tick as a regular tick, the backend could provide a **`notick`** instruction for the user to include as an additional event in each of these immediate lines.

3.6. Definition continuation lines

Additional event definitions for a definition line or immediate line may be placed on one or more *definition continuation lines*. A definition continuation line begins with at least one blank, and contains one or more event definitions separated by semicolons. It must follow a preceding definition line, immediate line, or definition continuation line, each of which must have terminated with a trailing "`;`". Interspersed comment lines containing only blanks and "`;`"s are also allowed. Note that a definition continuation line would be treated as a comment were it not for the trailing "`;`" on the preceding nonblank line. The last event definition on the last definition continuation line may be optionally

terminated by a semicolon *only* if the next nonblank line begins in column one. An individual event definition (an instruction name and its parameters) may *not* be split over several lines, putting some parameters on one line and some on another.

In Figure 4, note that a "**;**" may be placed after **instruc7**, since the next nonblank line starts in column one. In addition, if the "**;**" after **instruc4** were missing, then the definition would terminate on that line and the lines beginning with **instruc5** and **instruc7** would be parsed as comments, since the line containing **instruc4** would not have ended with a "**;**". This would occur even if there were a ";" immediately preceding **instruc5** on its line.

3.7. Super tick set lines

A *super tick set line* contains a "**#**" in column one, followed by a positive number. The length of super execution and continuation characters is set to that number of ticks. The default *super tick length* is 10.

3.8. Execution lines

An *execution line* begins with an event name. The event name must have been defined by an earlier definition line. The columns between the end of the event name and the ninth column (the first tab stop) must be blank. Each successive column in an execution line (starting at column nine) normally corresponds to a tick during which an event might be executed. Five kinds of characters may appear in a tick's column: *blank*, *execution character*, *continuation character*, *super execution character*, and *super continuation character*.

An event's execution is specified by an execution or super execution character optionally followed by continuation and super continuation characters. An event begins with an execution or super execution character and terminates with the last continuation or super continuation character before the next execution or super execution character. The continuation characters associated with an event may be broken over one or more execution lines following the line containing its execution character. The event is suspended during blanks interspersed among its execution and continuation characters.

```
% event1 instruc1 parm1; instruc2 parm1 parm2 parm3; instruc3 parm1

% event2 instruc1 parm1 parm2; instruc2; instruc3 parm1; instruc4;
        instruc5;instruc6 parm1 parm2 parm3;

        instruc7

% event3 instruc1 parm1; instruc2
```

Figure 4. Multiple event definitions.

3.8.1. Blank

The event is not executed during the tick.

3.8.2. Execution character

A "|" executes the first tick's worth of an event associated with the *execution line*.

3.8.3. Continuation character

A "-" executes the next tick of the event started by the "|" or "#" that was most recently encountered in an execution line with this event name.

3.8.4. Super execution character

A "#" acts like a "|" followed by enough "-"s to equal the current super tick length.

3.8.5. Super continuation character

A "=" acts like as many "-"s as are needed to equal the current super tick length.

3.9. Staves

A *staff* consists of a number of execution lines separated only by nonblank comment lines. All execution lines in a staff are incrementally executed in parallel, one tick (column) at a time. This pseudo-parallelism is accomplished by executing a tick's worth of each execution line in the order in which the lines appear in the staff. If more than one event is associated with an execution line, then they are executed, in the order in which they appeared in their definition line, before the events for the next execution line are executed. The length of a staff is that of its longest execution line. Thus, each execution line acts as if it were padded with enough blanks to make it equal the length of the staff's longest execution line. A blank comment (consisting solely of 0 or more blanks) will terminate a staff. Nonblank comments may be embedded in a staff without terminating it. A staff may contain at most one execution line with a given event name.

If a super execution or super continuation character appears in a column of a staff, then all blanks, regular execution characters, and continuation characters in that column of the staff are treated as if they were right-padded with enough blanks to equal the super tick length.

4. DIAL Programmer's Reference Manual

4.1. Overview

The instructions that a DIAL script can execute are implemented by one or more backend modules prepared by the programmer in accordance with the format specified below. All backend files that use dial should include the line

```
#include "dial.h"
```

There are only three entry points in the frontend:

DIALinit (backend)	Initializes dial to use a backend
DIALrun (script)	Runs a script
DIALtrace (flag)	Toggles execution trace output

The following sections describe the frontend entry points and data structures needed to construct a backend and run scripts that use it.

4.2. Frontend Calls

```
DIALbackend DIALinit (backend)
      DIALbackend backend;
```

DIALinit is called to initialize DIAL for a particular backend (defined by the **DIALbackend** data structure described below). It must be called before the first call to **DIALrun**. It should not be called from any of the programmer-defined routines called by **DIALrun**. **DIALinit** returns the previous **DIALbackend**. Note that it is only necessary to call **DIALinit** again in order to replace the old backend with a new one.

```
int DIALrun (script)
      char * script;
```

script is the filename of a DIAL script to be run. **DIALrun** may be called for as many scripts as desired after each invocation of **DIALinit**. **DIALrun** returns 0 if successful, nonzero if an error occurred.

```
int DIALtrace (flag)
      int flag;
```

An execution trace is output if **flag** is nonzero, suppressed otherwise (the default). **DIALtrace** returns the previous value of **flag**. **DIALtrace** may be called at any time. Many backends provide a **trace** instruction that invokes **DIALtrace** and also controls backend-specific tracing code. If **DIALtrace** is called before **DIALinit**, then the trace will include a listing of the backend contents.

4.3. Backend Data Structures and Entry Points

The user's program may be linked with one or more backends, each of which defines a separate instruction set. A backend contains a number of routines and data structures whose addresses DIAL learns about through the **DIALbackend** data structure passed to **DIALinit**. All of the data structures described below are defined in **dial.h**.

4.3.1. DIALbackend

```
typedef struct DIALbackend {              /* Dial backend          */
        DIALinstruction * instruction;    /* -> Instruction array */
        int num_instructions;             /* # of instructions     */
        int (*init_script) ();            /* script init routine   */
        int (*end_script) ();             /* script end routine    */
        int (*end_tick) ();               /* tick end routine      */
} DIALbackend;
```

A **DIALbackend** defines a DIAL backend. It contains the following elements:

instruction The address of an array containing one entry for each **DIALinstruction** (described below). The array must be sorted in alphabetically ascending order by the **name** field.

num_instructions The number of instructions in the **instruction** array.

init_script () Called before each script is executed, it performs whatever initialization the backend needs. It must return 0 if initialization was successful, nonzero otherwise (in which case execution of **DIALrun** terminates).

end_script () Called after each script is executed (even after error termination). It performs whatever cleanup the backend needs and must return 0 for success, nonzero otherwise.

end_tick () Called at the end of each tick. It would typically send the instructions that accomplish the animation. It must return 0 if the tick was successful, nonzero if an error occurred (in which case execution of **DIALrun** terminates).

4.3.2. Backend Routine Return Codes

Backend routines (**init_script**, **end_script**, **end_tick**, and the backend instruction procedures described below) all indicate successful execution by returning a zero return code. If any backend routine returns a nonzero return code, then **DIALrun** returns immediately (calling **end_script** first, if the nonzero return code was not returned from **end_script**). If the return code is positive, then **DIALrun** prints an appropriate error message and returns a positive return code (not necessarily that returned by the backend routine). This is the usual way to terminate a backend routine. If the return code is negative then **DIALrun** treats this as a "soft error." In this case, no error message will be printed and **DIALrun** will return the same negative return code returned by the backend routine that caused the soft error. The soft error facility makes it possible for any backend routine to terminate **DIALrun** quickly and silently, and to indicate the reason for termination by an appropriate return code. The backend must make its own arrangements if additional information related to the termination is to be made available to **DIALrun**'s caller when a soft error occurs.

4.3.3. DIALinstruction

```
typedef struct DIALinstruction { /* Dial instruction            */
        char * name;             /* Instruction name            */
        int (*proc) ();          /* Procedure to verify/execute */
        char * parm_type;        /* Parameter types             */
} DIALinstruction;
```

A **DIALinstruction** contains:

name The character string equivalent of the instruction name,

proc () The address of the procedure that will be called to execute the instruction and verify its parameters. A procedure whose address is assigned to **proc** may be declared static if it is included in the backend file.

parm_type A coded string indicating the number (by its strlen) and type of parameters to be parsed for and passed to the instruction. The type of the *n*th parameter is determined by the *n*th character of **parm_type**, which must be one of:

- **i** Integer – *atoi()* format or C hexadecimal constant format (e.g., **0x1b2f**).
- **f** Float – *atof()* format.
- **s** String – must be bracketed by double quotes ("). A double quote may be included if it is preceded by a backslash (\), but the backslash will remain in the string. Multiline strings (i.e., with embedded newlines) are not allowed. Strings are returned *exactly* as they occur in the script, null terminated and without their bracketing double quotes.

Both **proc** and **parm_type** may be changed during execution, allowing the routine and number/type of parameters associated with an instruction to be altered. An instruction's procedure may have up to **DIAL_MAX_PARMS** parameters.

4.3.4. Backend Instruction Procedures

```
int proc (done_ticks, total_ticks, info, op)
      int done_ticks;
      int total_ticks;
      DIALevent_info * info;
      int op;
```

Each instruction procedure is called to verify its parameters or to execute its share of the current tick's worth of the event. It is passed:

done_ticks The number of ticks done so far (1 for the first tick).

total_ticks The total number of ticks for the event.

info A pointer to the event's **DIALevent_info**.

op Either **DIAL_VERIFY** or **DIAL_EXECUTE**. If it is **DIAL_VERIFY** then the parameters in **DIALevent_info** are to

be verified for correctness (and fixed if necessary), but no execution is to be done. This occurs only once when the event is defined. If it is **DIAL_EXECUTE**, then the event should be executed for the current tick.

The procedure's return code is checked whenever it is called and **DIALrun**'s execution will be terminated if the return code is nonzero. Like all other backend routines, negative return codes are treated as defining "soft errors" (see the Section 4.3.2 above on **Backend Routine Return Codes.**)

4.3.5. DIALparm and DIALevent_info

```
typedef union DIALparm          /* Parameter                    */
{
        int i;                  /* Integer parm                 */
        float f;                /* Float parm                   */
        char * s;               /* String parm (-> script)      */
} DIALparm;

typedef struct DIALevent_info   /* Event's execution info       */
{
        int instruction_index;  /* DIALinstruction index        */
        int parm_count;         /* Number of parms              */
        DIALparm parm           /* Instruction's parms          */
           [DIAL_MAX_PARMS];
} DIALevent_info;
```

A **DIALevent_info** is passed to each instruction's procedure. It contains:

instruction_index The index of the instruction in the backend's **instruction** array. This allows multiple instructions to share the same instruction procedure, with the correct instruction distinguished by its index.

parm_count The actual number of parameters parsed by the frontend. It is never more than the number provided for in the **parm_type** entry in the instruction's **DIALinstruction.**, thus allowing trailing parameters to be made optional. The instruction's routine may substitute default values for missing parameters or treat them as an error.

parm The instruction's parameters. The relevant member of each **DIALparm** union is determined by the corresponding character ("**i**", "**f**", or "**s**") of the instruction's **parm_type** entry in its **DIALinstruction**.

All values in the **DIALevent_info** may be freely modified by the programmer, for example to set default values for optional parameters that were not passed. In the case of string parameters, both the address in the **parm** union and the actual string may be altered. If the original string is modified in place, then only characters before its terminating null may be changed. All programmer changes will be retained until the event is redefined.

4.3.6. Building a DIALbackend Data Structure

The following example shows one way to build a **DIALbackend** data structure:

```
static int inst_first ();        /* Instruction procedures        */
static int inst_second ();
static int inst_third ();


static
DIALinstruction inst [] = {      /* Array of dial instructions    */
        "first",        inst_first,             "s",
        "second",       inst_second,            "",
        "third",        inst_third,             "ssfii"
};


static int init ();              /* Script init routine           */
static int end ();               /* Script end routine            */
static int tick ();              /* Tick end routine              */


static
DIALbackend back = {             /* Dial backend                  */
        inst,
        sizeof (inst) / sizeof (DIALinstruction),
        init,
        end,
        tick
};
```

References

FEIN82a Feiner, S., Nagy, S., and van Dam, A. An experimental system for creating and presenting interactive graphical documents. *ACM Transactions on Graphics* 1:1, January 1982, 59-77.

FEIN82b Feiner, S., Salesin, D., and Banchoff, T. DIAL: A diagrammatic animation language. *IEEE Computer Graphics and Applications*, 2:7, September 1982, 43-54.