Inter-Office Memorandum

To	Cedar Interest	Date	January 15, 1981
From	Dan Swinehart	Location	Palo Alto
Subject	Cedar Terminal Input Facilities	Organization	CSL

XEROX

Filed on: [Ivy]<CedarDocs>User>InscriptImplementation.memo
[Ivy]<CedarDocs>User>InscriptImplAppendix.memo
Reference: [Ivy]<CedarDocs>User>InscriptDesign.memo (what I *intended* to do)

These will currently all be found on [Ivy]<Swinehart>Inscript>6.0>. They have also been released as part of Cedar.bcd.

Definitions files:
ClassInstream.bcd,
ClassIncreek.bcd,
Interminal.bcd,
Intime.bcd
InDiag.bcd
InOS.bcd
Implementation:
Instream.bcd, exports above interfaces, includes symbols
Instream.config
other source files, configs, etc.

Introduction

This memo describes the existing facilities for obtaining user terminal input. The most accurate interface description can be found in the cited definitions files; this discussion is intended to present both the main ideas and the details, but it may lag behind the truth from time to time.

User information is recorded in a file called an ***Inscript***. The inscript conceptually contains a time-stamped record of every activity, or *action*, the user has ever performed at the terminal. In reality, old information may be removed from this record, and it is possible that during some intervals not all user input is recorded (e.g., the detailed trajectories of the mouse.)

Most clients will obtain information from the inscript by creating an ***Instream***, which provides quite a bit of help with the interpretation of the user's activities -- primarily filtering and code-conversion assistance. Any number of instreams can be created, each of which may be examining different epochs in the input history.

Clients that wish to deal more directly with the recorded actions, perhaps to create a high-level input stream with different behavior from the **Instream** defined here, may create a lower-level stream-like object called an ***Increek***. The **Increek** provides operations for time-based positioning within the **Inscript**, and for examining Inscript actions.

Instreams

An **Instream** provides a stream of ***Events***, currently confined to keyboard, keyset, and mouse button (henceforth shortened to "key") activations. **Instream** provisions for tracking the mouse in the absence of key activations are TBD, although complete facilities exist at the **Increek** level. Not all user activities, or ***Actions***, result in **Events**, and some **Events** are not the immediate result of user activity. The **Instream** client can control to some extent which **Actions** result in **Events**.

The client can choose to obtain and examine **Events** directly from the **Instream**, or to obtain derivative data types based on the type of **Event** (keyboard characters, mouse button clicks, etc.) After obtaining an **Event** or a derivative, the client can also examine the state of the **Instream** to obtain time stamp information, the position of the mouse, or the state of various keys (shift, lock, control, etc., or any other key for that matter) at the time of the event.

Data Types

What follows is a discussion of the more important data types used or viewed by **Instream** and **Increek** clients.

From ClassInstream

```
ClassInstream.Instream: TYPE = REF InstreamObject;
```

This is an object that implements the operations described below.

```
ClassInstream.Device: TYPE = {nullDevice, char, button, paddle, motion};
```

An **Instream** delivers **Events** to its clients. A field in each **Event** identifies which device generated it (keyboard, mouse, mouse motion only, or keyset), and what kind of activity caused it. Motion events are not yet implemented.

```
ClassInstream.Cause: TYPE = MACHINE DEPENDENT {none(0),
    strobeDown(1), strobeUp(2), heldDown(4), heldUp(10B), canBeChord(20B)};

ClassInstream.Causes: TYPE = -- Powerset of -- Cause;
```

The **Instream** client uses values of type **Causes** to specify which of the user activities are of interest for each of the three devices (there are standard defaults.) **strobeDown** means that an **Event** will occur whenever a key is depressed. **strobeUp** means that an **Event** will accompany the release of a key on the selected device. **heldDown** and **heldUp** will occur only after a key has been depressed (released) for a specified duration, in the absence of additional keystrokes. This is in support of multiple click (and later "typeamatic") **Event**s, described further in a later section. For simple situations one merely specifies **strobeDown**, **strobeUp**, or both.

Events also contain **Cause** fields, to indicate to the client which kind of activation occurred.

Cause has its values explicitly specified so that values can be **OR**ed together to specify multiple activations. There are "powerset" functions, described below, for creating the desired configurations.

```
ClassInstream.Event: TYPE = RECORD [
   device: Device,
   keyName: Interminal.KeyName,
   cause: Cause,
   clicks: CARDINAL
   ];
```

The **Event** records which device was involved, what kind of activation occurred, and which key was depressed (using names assigned from a space containing them all). Each **Event** also contains a **clicks** field, described further in the *Clicks* section. The stream stores additional information about the current state of all the input devices just after the **Event** occurred. This state includes the current activation state of *all* of the keys, the mouse position at the time of the **Event,** the time of the **Event**,

the "clicks" information, and a **chord** field that has collected all key depressions since the last time all keys were undepressed. The **Instream** provides operations for obtaining these values.

```
ClassInstream.EventTime: TYPE=Intime.EventTime;

Intime.EventTime: --(effectively)-- TYPE[3];
```

This is not really an exported type at present, being fully specified in the definitions file. But its internal fields are not meaningful to clients. An **EventTime** represents the number of milliseconds since some specific time early in the century. Functions exist (in the **Intime** interface) for converting the present time to an **EventTime,** for performing simple arithmetic and comparisons on these times, etc. **EventTimes** may be used to denote positions within the **Instream.**

```
ClassInstream.MousePosition: TYPE = Interminal.MousePosition;

Interminal.MousePosition: TYPE = MACHINE DEPENDENT RECORD [
      mouseX: CARDINAL, mouseY: CARDINAL];
```

These are expressed in Alto screen coordinates, at present. Expect these representations to change in order to adapt to Cedar graphics' more abstract view of things.

```
ClassInstream.PosResult: TYPE = ClassIncreek.PosResult;

ClassIncreek.PosResult: TYPE = {tooEarly, tooLate, onTime};
```

Describes the nature of the results of some of the **Instream** operations below.

```
ClassInstream.MsTicks: TYPE = Intime.MsTicks;

Intime.MsTicks: TYPE = Process.Milliseconds -- ... = CARDINAL --;
```

Describes the nature of the results of some of the **Instream** operations below.

From Interminal

```
Interminal.KeyView: TYPE = {keyNames, keyFields, words, bits};
Interminal.KeyState: TYPE = MACHINE DEPENDENT RECORD [
   SELECT OVERLAID KeyView FROM
     bits => [bits: PACKED ARRAY KeyName OF updown],
     words => [words: KeyArray],
     keyNames => [keyNames: KeyNames],
     keyFields=> [keyFields: KeyFields],
     ENDCASE];

Interminal.KeyName: TYPE = {
     x0, x1, ..., x7,
     Keyset1,Keyset2,...,Keyset5,
     Red,Blue,Yellow,
     Five,...,U,V,
     ...
     Lock,Space,...,FR5};
```

There are a number of useful ways to look at a representation of the keyboard state; these are the ones that have been chosen for Cedar input. The bits array can be indexed by the **KeyName** values, which are also the values that are stored in **Actions** and **Events**. The **keyNames** variant lets you treat the key state as a record of named bits. **keyFields** breaks the state up into smaller records, by device, once you know which device you're interested in. The client of **Instreams** and **Events** will normally require only **KeyNames**, but may need to access other values to examine the control, shift, shift lock keys, or other "non-strobing" situations in efficient ways.

```
Interminal.PaddleKeyName, KeyArray, KeyNames, KeyFields, Buttons,
        ButtonNames, Paddles, PaddleNames: TYPE = ...;
```

These types are used to implement the ones described above. Every implementor of an **Instream** or **Increek** client should read through them and pick out the types that seem relevant to the particular way the client code is going to use the package; the implementor will probably settle on a fairly small number of them.

```
Interminal.Spare1: CHARACTER=LOOPHOLE[201B];
Interminal.Spare2: CHARACTER=LOOPHOLE[202B];
Interminal.Spare3: CHARACTER=LOOPHOLE[203B];
Interminal.ShiftSpare1: CHARACTER=LOOPHOLE[204B];
Interminal.ShiftSpare2: CHARACTER=LOOPHOLE[205B];
Interminal.ShiftSpare3: CHARACTER=LOOPHOLE[206B];
```

These definitions provide representations in the "Ascii" domain for the three spare keys and their shifted versions. They were arbitrarily chosen.

Procedures and Operations

Some of these functions are currently implemented as **INLINE** procedures, and are thus tied to a particular implementation of the Instream data record type. These will be changed if necessary.

```
ClassInstream.EventIncorrect: ERROR[e: Event];

ClassInstream.EventVanished: ERROR[eT: Intime.EventTime];

ClassInstream.NewStdInstream: PROCEDURE RETURNS [ClassInstream.Instream];
```

Creates an **Instream** on the standard **Inscript** that is filled with user terminal actions. It is possible to produce Instream implementations that obtains actions from a different source, in which case this function would not be used to create them. The result is an **Instream** object. It will have been positioned at the end of (the latest point in) the **Inscript** (corresponding to "now".)

The implementation does not enforce a limit on the number of **Instreams** that may be created to examine the same **Inscript**. This design will make more sense if at least two Instreams get created by somebody in some set of Cedar applications;.

```
ClassInstream.GetEvent: PROCEDURE[self: ClassInstream.Instream] RETURNS [e:
    ClassInstream.Event];
```

Returns the next **Event** from the stream that satisfies the client's specifications (see below.) If the stream is currently positioned at the end of the **Inscript**, this operation will wait as necessary until actions constituting an acceptable **Event** have occurred. The client program should not be able to detect a difference between waiting and non-waiting situations, unless the client is also keeping track of real time.

Raises **ClassInstream.EventVanished** if the **Events** denoted by the current **Instream** position have disappeared to make room for newer ones. This will only happen if the Instream is reviewing ancient history. The **EventTime** parameter denotes the time of the earliest **Action** remaining in the **Inscript**.

```
ClassInstream.GetChar: PROCEDURE[self: ClassInstream.Instream, event:
    ClassInstream.Event_nilEvent] RETURNS [c: CHARACTER];
```

If **event** is **nilEvent** (not supplied in the call), an **event** is obtained from the **Instream**. If the **event** represents a keyboard keystroke, its Ascii code is returned. Otherwise **GetChar** raises **ClassInstream.EventIncorrect**, with the **Event** as a parameter. The intent is that a "character loop" can call **GetChar** repeatedly, acting on the incoming characters until an event occurs that is not a character event. Alternatively, the client can obtain an **Event** using **GetEvent**, determine that it is a char event, then obtain the corresponding Ascii code using **GetChar**.

There may never be any "character loop" types of applications, in which case we should simplify this interface. The original plan also called for leaving the stream positioned at its pre-call location when **EventIncorrect** was raised, returning just the event type as an error value, so that the client could retry the operation. The current implementation does not do this, on the grounds of expense. The plan also called for functions to reposition the stream by small integral numbers of **Events** in either direction, and other such (hopefully) nonsense.

```
ClassInstream.GetPaddle: PROCEDURE[self: ClassInstream.Instream, event:
    ClassInstream.Event_nilEvent] RETURNS [e: ClassInstream.Event];

ClassInstream.GetButton: PROCEDURE[self: ClassInstream.Instream, event:
    ClassInstream.Event_nilEvent] RETURNS [e: ClassInstream.Event];
```

These are like **GetChar**, but directed at the other input devices. Each returns the full **Event** that **GetEvent** would return, but only if the **Event** is caused by the requested device; otherwise, they raise **EventIncorrect**.

```
ClassInstream.GetPaddleName: PROCEDURE[self: ClassInstream.Instream, keyName:
    Interminal.KeyName]
    RETURNS [name: Interminal.PaddleName];

ClassInstream.GetButtonName: PROCEDURE[self: ClassInstream.Instream, keyName:
    Interminal.KeyName]
    RETURNS [name: Interminal.ButtonName];
```

These apply appropriate offsets to the **keyName** values to produce values from smaller ranges specific to their devices. They should, but do not, complain when they produce values that are out of range.

```
ClassInstream.GetMousePosition: PROCEDURE[self: ClassInstream.Instream]
    RETURNS [mP: ClassInstream.MousePosition];
```

Returns the mouse position that obtained just after the (occurrence of the) ***Last Action*** (in the most recently obtained **Event**).

```
ClassInstream.GetEventTime: PROCEDURE[self: ClassInstream.Instream]
    RETURNS [eT: ClassInstream.EventTime];
```

Returns the time at which the **Last Action** occurred.

```
ClassInstream.GetCurrentTime: PROCEDURE[self: ClassInstream.Instream]
    RETURNS [eT: ClassInstream.EventTime];
```

Defines "now" so that you can come back to it later. There are a number of functions defined in **Intime** that allow for calculations on and comparisons of **EventTimes**. This function is not dependent on **Instream** position.

```
ClassInstream.GetStateOfKey: PROCEDURE[self: ClassInstream.Instream, keyName:
    Interminal.KeyName]
    RETURNS [state: Interminal.updown];
```

Returns the state of the selected key just after the **Last Action**.

```
ClassInstream.GetKeyState: PROCEDURE[self: ClassInstream.Instream]
    RETURNS [keyState: Interminal.KeyState];

ClassInstream.GetChordState: PROCEDURE[self: ClassInstream.Instream]
    RETURNS [chordState: Interminal.KeyState];
```

GetKeyState returns the entire state of the key devices just after the **Last Action**. **GetChordState** is similar, returning a **KeyState** representing all the the keys that have been depressed since they were all up (**downCount** was last zero).

ClassInstream.GetDownCount: PROCEDURE[self: ClassInstream.Instream, device: ClassInstream.Device_nullDevice] RETURNS [downCount: CARDINAL];

Returns the number of keys that were depressed just after the **Last Action**, for the specified device. If **device** is omitted or is **nullDevice**, returns the number of keys were depressed for all devices (this is somewhat more efficient.)

ClassInstream.GetChord: PROCEDURE[self: ClassInstream.Instream] RETURNS [s: keyFields Interminal.KeyState];

This procedure is not yet implemented, nor is it fully designed. Its intent is to provide a high-level method for obtaining the complete value of the "next chord", once the client detects that a chord-like activity is in progress.

ClassInstream.SetAtEarliest: PROCEDURE[self: ClassInstream.Instream];

Positions the stream preceding the earliest known action. For reviewing the history of the universe. In some implementations (this one, for example), one is well-advised to proceed with dispatch to examine these earlier actions before they go away to make room for new ones; if that occurs, the **Get**... routines will raise a signal, but this aspect of the implementation is not worked out very well yet.

ClassInstream.SetAtLatest: PROCEDURE[self: ClassInstream.Instream];

Positions the stream to "now".

ClassInstream.SetAtTime: PROCEDURE[self: ClassInstream.Instream, eventTime: ClassInstream.EventTime] RETURNS [pR: PosResult];

Positions the stream to the first point at or following the specified time. The **PosResult** return value indicates whether the specified time preceeds the earliest known action, follows the last known action, or lies somewhere within the known history. In any case, the stream is positioned at the nearest approximation to the selected time.

SetEventSpecifications: PROCEDURE[self: ClassInstream.Instream, device: ClassInstream.Device, causes: ClassInstream.Causes, clickTime: ClassInstream.MsTicks_0];

device is used to specify the device, causes the **Actions** that are to result in **Events** for that device, and **clickTime** the timeout value, in milliseconds, that will subsequently be used to control "click" computations *for all the devices* (see the section on multiple clicks). The initial defaults for the devices are:

[device: char, causes: strobeDown]
[device: button, causes: {strobeDown, strobeUp}] (e.g., PowerSet[sD, sU])
[device: paddle, causes: {strobeDown, strobeUp}]
[device: motion, causes: <not applicable, not implemented.>]

This produces **Events** only when keyboard keys are depressed, and when mouse/keyset keys are activated one way or the other.

Some Inlines defined in ClassInstream

ClassInstream.PowerSet: PROCEDURE [e1, e2, e3, e4: ClassInstream.Cause_none] RETURNS [ClassInstream.Causes] = ...;

ClassInstream.In: PROCEDURE[candidate: ClassInstream.Cause, target: ClassInstream.Causes] RETURNS [BOOLEAN] = = ...;

PowerSet[...] creates a **Causes** value, given two or more **Cause** or **Causes** values. The result represents a set of **Causes**. What really happens is that the bits are OR'ed together; **Cause** is defined so that the named values are powers of 2.

In[...] determines whether the **candidate** is a member of the set of **Causes** in the **target**. If the candidate is a **Causes** value, all of its members must be in the **target**. (**TRUE** iff **LOGAND[candidate, target] = candidate**.)

These functions use inline definitions available in Powerset.Mesa/bcd, redefining their input and output types.

Multiple "Clicks"

Many user interfaces, including *Tioga's*, use the concept of multiple activations of the mouse, typically within a given time and without excess motion of the mouse, to increase the number of interpretations that can be assigned to mouse buttons. In the Inscript world, with its emphasis on the recording of actions, and the ability to respond to them more than once, and perhaps long after they occur, it seemed prudent to capture as much of the complexity of multi-click activities as possible at the Instream level of the user input facilities.

The current interpretation is based almost entirely on Tioga's needs. Most of the extensions that come to mind could be readily accommodated.

The client uses **SetEventSpecifications** to specify, for each device, what kinds of events are interesting. If either **heldDown** or **heldUp** is specified, the client must also supply a time, in milliseconds, that is used as follows.

Suppose that all keys are "up", and have been for some time. The client has specified that, on the mouse, only **heldDown** and **heldUp** Events are interesting (at present, specifying one is equivalent to specifying both -- an implementation expedience), with a **clickTime** of 100 ms. The user now depresses a mouse button. The Instream would not immediately interpret this **Action** as an **Event**, but would "start a timer" to expire in 100 ms. If the user moves the mouse "too far" (currently a constant -- what should it be?) within that 100 ms., or depresses or releases some other key, or does not perform any more **Actions** within the 100 ms., a heldDown event results. If the user releases the button within the 100 ms., the timer is restarted, and no **event** results. When the user finally waits long enough, moves far enough, or activates some other key, a **heldDown** or **heldUp event** results.

If **strobeDown** and/or **strobeUp** had also been specified in the above example, each of the actual key activations would also have resulted in an **Event**; these additional events would have been properly interleaved with exactly the same held-style **Events** that occurred in the example.

If **held... events** have been specified, each **event** will include, in the **clickCount** field, the number of key activations (down or up), including the one that resulted in the event, that have occurred within the same multi-click sequence. Otherwise, the **clickCount** field will always contain a 1 or something.

Note that all of these time-related operations use the times that are recorded in the **Inscript**; the semantics of the click activities are independent of the celerity with which the client is examining the **Inscript**. However, the amount of real time that a given call on, say, **GetEvent** will require *is* dependent on whether the stream is positioned at "end of script."

There may well be other useful things to say about this implementation; what are they?

Increeks

An **Increek** is a lower-level object that can be used to gain access to an **Inscript**. The **Instream** implementation, in fact, simply provides interpretations for collections of low-level individual user **Actions** which are extracted using **Increeks**. **Increek** operations largely parallel the **Instream** operations. Details and discussion of the **Increek** level follow.

Data Types

From ClassIncreek

One obtains **Actions** by invoking operations in an object called an **Increek**. The kind field of the **Action** indicates what happened, and the other fields provide the details.

```
ClassIncreek.Increek: TYPE = REF IncreekObject;

ClassIncreek.ActionKind: TYPE = {deltaEventTime, eventTime, deltaMouseX,
    deltaMouseY, mousePosition, keyDown,  keyUp, keyStillDown};

ClassIncreek.Action: TYPE = LONG POINTER TO ClassIncreek.ActionBody;

ClassIncreek.ActionBody: TYPE = RECORD [
    deltaDeltaTime: Intime.DeltaDeltaTime _ 0,
    contents: SELECT kind: ClassIncreek.ActionKind FROM
        deltaEventTime => [value: Intime.DeltaTime _ NULL],
        deltaMouseX, deltaMouseY =>
            [value: ClassIncreek.DeltaMouseValue _ NULL],
        keyDown, keyStillDown, keyUp =>
            [value: Interminal.KeyName _ NULL],
        eventTime => [eventTime: Intime.EventTime _ NULL],
        mousePosition => [mousePosition: Interminal.MousePosition _ NULL],
        ENDCASE
    ];
```

If the time between **Actions** is short enough, the time difference can be recorded in the **deltaDeltaTime** field of the next **Action**. Otherwise it will appear as a separate **Action** preceding the "real" one, either as a **deltaEventTime Action** if the interval is not too long, or as a full **eventTime** value. Similarly, mouse motions are recorded either as small incremental **mouseX** and **mouseY Actions** or as full **mousePosition** values. The **keyUp** and **keyDown Actions** correspond to actual user keyboard activities.

keyStillDown is used to record efficiently the entire keyboard state at the beginning of each "session", and at the beginning of each inscript page; one **Action** appears for each key that is "still down", typically zero, one, or two. This allows the state to be recreated efficiently without scanning the entire history. The full time and full mouse position are also inserted into the inscript at these times. **keyStillDown[value: allUp]** should be interpreted as a request to clear the state to indicate no depressed keys.

As with an **Instream**, an **Increek** defines a position within its **Inscript**. This position represents an **EventTime** at which the input devices were in a particular state, defined by the type **InscriptPosition** (**ViewPosition** was intended as a **READONLY** version of this type, but I couldn't get it to work right.)

```
ClassIncreek.InscriptPosition: TYPE = REF ClassIncreek.InscriptPositionBody;

ClassIncreek.ViewPosition: TYPE = REF -- can't get READONLY to work right --
    ClassIncreek.InscriptPositionBody;
```

```
ClassIncreek.InscriptPositionBody: TYPE = RECORD [
    -- location in inscript file --
    inscript: PRIVATE ClassInscript.Inscript, -- for releasing
    inscriptPage: PRIVATE ClassInscript.InscriptPageDescriptor,

    -- absolute state at that point --
    eventTime: Intime.EventTime _ NULL,
    mousePosition: Interminal.MousePosition _ NULL,
    keyState: Interminal.keyState _ NULL,
    chordState: Interminal.keyState _ NULL,
    downCount: INTEGER _ 0
    ];
```

chordState is cleared to **allUp** whenever **keyState** is about to leave the **allUp** condition; **keyDown** and **keyStillDown** events cause the corresponding **chordState** bits to be set, but **keyUp** events do not effect **chordState**. Probably the shift lock key should not participate in the **chordState**.

PosResult describes the nature of the results of some of the **Increek** operations below. **Acceptance** is a client-provided parameter to **GetAction**. **DeltaTime** and **DeltaDeltaTime** are time values of differing lengths, in units corresponding to the process-scheduling "tick" interval. Intervals at this resolution can be represented in fewer bits. Unless the client examines these **Action** fields directly, it will not have to deal with these units, since most of the time-related functions convert these times back to millisecond-resolution units. **MousePosition**, **KeyState**, etc., are as defined in the *Instreams* section. **DeltaMouseValue** is a short, incremental version of the components of **MousePosition. WaitMode** determines the timeout behavior of the **GetAction** operation.

```
ClassIncreek.PosResult: TYPE = {tooEarly, tooLate, onTime};

ClassIncreek.Acceptance: TYPE = {clicks, clicksAndMotion, all};

ClassIntime.DeltaTime: TYPE = ... CARDINAL [0..256);

ClassIntime.DeltaDeltaTime: TYPE = DeltaTime [0..32);

ClassIncreek.DeltaMouseValue: TYPE = [-128..128);

ClassInscript.WaitMode: TYPE = {forever, dontWait, timed};
```

Procedures and Operations

```
ClassIncreek.IncreekError: ERROR[code: ClassIncreek.IncreekErrorCode];

ClassIncreek.IncreekErrorCode: TYPE = {
    outOfBounds -- position no longer valid during ReadAction
    };
```

NewStdIncreek provides the standard **Increek** implementation. Its **Actions** are obtained from the standard **Inscript** implementation, which directly records user terminal events. If the **template** argument is **NIL**, the result is a new **Increek** on which the **SetAtLatest** operation has just been performed.

One can also call this function to obtain an **Increek** that is a copy of another **Increek**, the **template**; the copy represents another source of Actions, positioned at the same point in the script. This "produce copy" function should be an object operation.

The **CopyIncreek** operation is equivalent to **SetAtTime[self, GetTime[template]];**

```
ClassIncreek.NewStdIncreek: PROCEDURE[template: ClassIncreek.Increek_NIL] RETURNS
    [ClassIncreek.Increek];

ClassIncreek.Release: PROCEDURE [self: ClassIncreek.Increek] RETURNS [nilIncreek:
    ClassIncreek.Increek];

ClassIncreek.CopyIncreek: PROCEDURE [self: ClassIncreek.Increek, template:
    ClassIncreek.Increek];
```

The **SetAt...** functions in the **ClassInstream** interface parallel the ones provided here. Each of these functions positions the **Increek** such that the next **Action** is the earliest in the **Inscript** that occurred after the specified time. In the process of positioning the **Increek**, it updates the Increek's **ViewPosition** (state record) to represent the state of the terminal corresponding to the new position.

The **Get...Time** functions return the same values as the corresponding functions in the **ClassInstream** interface. **GetPositionFrom** returns the **ViewPosition** record describing the terminal state at the current position. It is provided in lieu of the slew of functions (as in **ClassInstream**) that would otherwise be required; the **ClassIncreek** interface is intended for use by system implementors to provide higher-level interfaces.

```
ClassIncreek.SetAtEarliest: PROCEDURE [self: ClassIncreek.Increek];

ClassIncreek.SetAtLatest: PROCEDURE [self: ClassIncreek.Increek];

ClassIncreek.SetAtTime: PROCEDURE [self: Increek, eventTime: Intime.EventTime]
    RETURNS [pR: ClassIncreek.PosResult];

ClassIncreek.GetTime: PROCEDURE [self: ClassIncreek.Increek]
    RETURNS [eT: ClassIncreek.EventTime];

ClassIncreek.GetCurrentTime: PROCEDURE [self: ClassIncreek.Increek]
    RETURNS [eT: ClassIncreek.EventTime];

ClassIncreek.GetPositionFrom: PROCEDURE [self: ClassIncreek.Increek]
    RETURNS [p: ClassIncreek.ViewPosition];
```

Finally, to examine the next action in the Inscript, use **GetAction**. It returns an **Action** that is *extremely* unsafe -- it is a **LONG POINTER** to an **ActionBody** whose data is guaranteed not to change only until the next call on **GetAction** or **SetAt..**. Copy it if you want it to last longer. The **waitMode** and **waitInterval** parameters allow for the implementation of higher level abstractions that involve timeouts. They operate based on the times *recorded in the file*, rather than the present real time. Thus, **GetAction** may return with a timeout indication immediately after the client calls it. If **waitMode** is forever, **GetAction** will not return until an **Action** has occurred; if **dontWait**, return is immediate unless additional **Actions** remain in the **Inscript**. If **waitMode** is **timed**, the next Action will be returned only if it occurred within **waitInterval** ticks following the preceding Action. Exception: if the Increek is at "end of file", GetChar will wait the indicated interval (in the absense of new Actions) before returning; and this interval begins at the *real time* of the call. These semantics are considered to be a compromise.

GetAction will raise **IncreekError[outOfBounds]** if the **Increek** is examining sufficiently old **Actions** that their storage must be released to make way for new ones.

```
ClassIncreek.GetAction: PROCEDURE [self: ClassIncreek.Increek, waitMode:
    ClassInscript.WaitMode _ forever, waitInterval: Intime.MsTicks _ 100,
    acceptance: ClassIncreek.Acceptance _ clicks]
    RETURNS [a: ClassIncreek.Action];
```

GetAction returns **NIL** if the timeout criteria hold or if it encounters an **Action** not requested by the acceptance parameter. This makes it very difficult, without rechecking the timeout criteria in the client, to determine why a **NIL** resulted; for some reason this state of affairs is acceptable to the current implementation, but it should probably be changed.

How to Use

The entire package is available as Instream.bcd, on the above-cited directory. The definitions files of interest to the client are also available there. It imports the **CWF** package rather than including it; all other imports (it is claimed) come from the system.

Start the system by creating an **Instream** via **ClassInstream.NewStdInstream** (or, if you're not using **Instreams**, **ClassIncreek.NewStdIncreek**. It will take several seconds and will create a file "Inscript.Inscript." This file is made to reflect "truth" every few seconds; subsequent "runs" will find the current position in this file.

Once the package has started, the standard Mesa keyboard/cursor packages have been disabled (but not necessarily removed from storage.) The <shift>SWAT, <ctrl>SWAT, and <ctrl><shift>SWAT functions still work, although they also cause Actions to be recorded, and will not work on replay.

Current Shortfalls

1. There's no way to reduce the number of actions placed into the inscript by the terminal code -- i.e., by eliminating or reducing the frequency of mouse motion actions, when not accompanied by clicks, at times when such actions are not of interest (most of the time.) Implementing this feature would not be without risk -- during type ahead situations, there might not be enough available information.

2. There is currently no event filtering based on mouse coordinates. This could be supplied elsewhere, but it was always intended to provide such facilities here. I believe that the clipping capabilities of Cedar graphics can be used to make this filtering very efficient and effective. Whether this is done at this level depends to some extent on how many independent processes end up "pulling." I'll discuss this further with relevant parties. Mouse coordinates are also currently expressed in screen units, rather than any device-independent manner. They clearly should be recorded this way, but may want to be presented to clients differently.

3. There is currently no version that records actions forever. This should perhaps be done, if at all, at the event level, after it is determined which actions are interesting. The current inscript is a circular buffer of (now a constant 100 Alto pages, soon to be variable) length, which can capture about a minute's worth of reasonable continuous activity on each page.

4. There is no specific mechanism included for changing cursor shape based on mouse position (and perhaps the state of the mouse buttons.) Is this needed? If so, I can treat it along with point 2.

5. Chord-fetching operations are perhaps still too primitive. **ClassInstream.GetChord** is not implemented, nor is it very well defined. **ClassInstream.GetChordState**, along with **ClassInstream.GetDownCount**, leave one with not much work remaining, however. If Shift lock is locked down, simple algorithms for dealing with chords will fail.

The interfaces described here should be sufficient for most clients of the Inscript system, except for those who desire to implement the interfaces for different sources, or using different filing methods. For the more adventurous, an appendix follows. It describes the Intime and Interminal interfaces in more detail, and discusses some aspects of the current implementation.

The Interminal Definitions

The definitions module **Interminal** supplies a set of data types that describe the "Alto" terminal input configuration: keyboard, keyset, and mouse. The **InterminalImpl** module, part of the current implementation, does not export anything to **Interminal**, so the names are in that sense coincidental. A later section discusses this implementation.

The intent of the **Interminal** types is indicated in the previous *Instreams* sections; the details are included here for completeness.

Data Types

```
Interminal.MousePosition: TYPE = MACHINE DEPENDENT RECORD [
    mouseX: INTEGER, mouseY: INTEGER];

Interminal.updown: TYPE = {down, up};

Interminal.KeyView: TYPE = {keyNames, keyFields, words, bits};

Interminal.KeyState: TYPE = MACHINE DEPENDENT RECORD [
    SELECT OVERLAID KeyView FROM
      bits => [bits: PACKED ARRAY Interminal.KeyName OF Interminal.updown],
      words => [words: Interminal.KeyArray],
      keyNames => [keyNames: Interminal.KeyNames],
      keyFields => [keyFields: Interminal.KeyFields],
      ENDCASE
    ];

Interminal.KeyName: TYPE = {
  x0, x1, x2, x3, x4, x5, x6, x7, Keyset1, Keyset2, Keyset3, Keyset4, Keyset5, Red, Blue, Yellow,
  Five, Four, Six, E, Seven, D, U, V, Zero, K, Dash, P, Slash, BackSlash, LF, BS,
  Three, Two, W, Q, S, A, Nine, I, X, O, L, Comma, Quote, RightBracket, Spare2, BW,
  One, ESC, TAB, F, Ctrl, C, J, B, Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3,
  R, T, G, Y, H, Eight, N, M, Lock, Space, LeftBracket, Equal, RightShift, Spare3, FL4, -- FR5, -- allUp
  };
```

This implementation has purloined the code for **FR5** -- "Cedar" terminals do not use it -- in order to obtain a code for the use of the implementation. The client may see this code, renamed **allUp**, as part of a **stillDown** action -- see the **Increeks** section.

The following **INLINE** functions are just pretty loopholes for the implementation's benefit.

```
Interminal.Kn: PROCEDURE [value: UNSPECIFIED] RETURNS [kN: KeyName] = INLINE {
    kN _ LOOPHOLE[value]};

Interminal.Kv: PROCEDURE [value: UNSPECIFIED] RETURNS [kV: CARDINAL] = INLINE {
    kV _ LOOPHOLE[value]};

Interminal.KbdKeyName: TYPE = KeyName [Five..allUp];

Interminal.ButtonKeyName: TYPE = KeyName [Red..Yellow];

Interminal.PaddleKeyName: TYPE = KeyName [Keyset1..Keyset5];

Interminal.KeyArray: TYPE = ARRAY [0..5) OF WORD;

Interminal.KeyNames: TYPE = MACHINE DEPENDENT RECORD [
```

```
    blank: [0..377B],
    Keyset1, Keyset2, Keyset3, Keyset4, Keyset5: updown, Red, Blue, Yellow: updown,
    Five, Four, Six, E, Seven, D, U, V, Zero, K, Dash, P, Slash, BackSlash, LF, BS: updown,
    Three, Two, W, Q, S, A, Nine, I, X, O, L, Comma, Quote, RightBracket, Spare2, BW: updown,
    One, ESC, TAB, F, Ctrl, C, J, B, Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3: updown,
    R, T, G, Y, H, Eight, N, M, Lock, Space, LeftBracket, Equal, RightShift, Spare3, FL4, -- FR5 -- allUp:
        updown
    ];

Interminal.KeyFields: TYPE = MACHINE DEPENDENT RECORD [
    mesaMemorialBlankField: [0..377B],
    paddles: Interminal.Paddles,
    buttons: Interminal.Buttons,
    keys: ARRAY [0..3] OF WORD];

Interminal.ButtonView: TYPE = {buttonChord, buttonNames, buttonValue};

Interminal.Buttons: TYPE = MACHINE DEPENDENT RECORD [
    SELECT OVERLAID ButtonView FROM
      buttonChord => [buttonChord: [0..10B)],
      buttonNames => [buttonNames: Interminal.ButtonNames],
      buttonValue => [buttonValue: Interminal.ButtonValue],
      ENDCASE
    ];

Interminal.ButtonNames: TYPE = MACHINE DEPENDENT RECORD
    [Red, Blue, Yellow: updown];
```

Several of the enumerated types that follow specify their implementation representations explicitly so that said representations can be powers of two. This allows them to be used in conjunction with the **PowerSet** functions, described with **Instreams**.

```
Interminal.ButtonName: TYPE = MACHINE DEPENDENT{Red(0), Blue(1), Yellow(2)};

Interminal.ButtonValue: TYPE = MACHINE DEPENDENT{
    RedYellowBlue(0), RedBlue(1), RedYellow(2), Red(3), BlueYellow(4),
    Blue(5), Yellow(6), None(7)};

Interminal.PaddleView: TYPE = {paddleChord, paddleNames, paddleValue};

Interminal.Paddles: TYPE = MACHINE DEPENDENT RECORD [
    SELECT OVERLAID PaddleView FROM
    paddleChord => [paddleChord: [0..40B)],
    paddleNames => [paddleNames: Interminal.PaddleNames],
    paddleValue => [paddleValue: Interminal.PaddleValue],
    ENDCASE];

Interminal.PaddleNames: TYPE = RECORD [
    Keyset1, Keyset2, Keyset3, Keyset4, Keyset5: updown];

Interminal.PaddleName: TYPE = MACHINE DEPENDENT{
    Keyset1(0), Keyset2(1), Keyset3(2), Keyset4(3), Keyset5(4)};

Interminal.PaddleValue: TYPE = MACHINE DEPENDENT{
    Keyset1(17B), Keyset2(27B), Keyset3(33B), Keyset4(35B), Keyset5(36B),
    None(37B)};
```

```
Interminal.allUp: KeyState;

Interminal.Spare1: CHARACTER=LOOPHOLE[201B];
Interminal.Spare2: CHARACTER=LOOPHOLE[202B];
Interminal.Spare3: CHARACTER=LOOPHOLE[203B];
Interminal.ShiftSpare1: CHARACTER=LOOPHOLE[204B];
Interminal.ShiftSpare2: CHARACTER=LOOPHOLE[205B];
Interminal.ShiftSpare3: CHARACTER=LOOPHOLE[206B];
```

Intime

Intime provides data types for representing **EventTime** values in a number of sizes; all represent times, or time increments, expressed in milliseconds. There are also some incremental time values expressed in a courser grain (the actual value stored in **Increek Actions**.) This interface also supplies a number of procedures for manipulating these various times and their incremental derivatives.

Data Types

```
Intime.MsTicks: TYPE = Process.Milliseconds;

Intime.MSTicks: TYPE = LONG CARDINAL; -- long milliseconds --

Intime.DeltaTicks: TYPE = Process.Ticks; -- probably 1/60 sec.

Intime.DeltaTime: TYPE = Intime.DeltaTicks [0..256);

Intime.maxDeltaTime: Intime.DeltaTime = LAST[Intime.DeltaTime];

Intime.msPerDeltaTick: Intime.MsTicks; -- converts between the representations

Intime.deltaTicksPerSecond: Intime.DeltaTicks;

Intime.DeltaDeltaTime: TYPE = Intime.DeltaTime [0..32);

Intime.maxDeltaDeltaTime: Intime.DeltaDeltaTime = LAST[Intime.DeltaDeltaTime];

Intime.Overlap: TYPE = {loShort, hiShort};
```

Client should think of **EventTime** as **TYPE[3].**

```
Intime.EventTime: TYPE = PRIVATE MACHINE DEPENDENT RECORD
    [
    SELECT OVERLAID Overlap FROM
      loShort => [lo: MsTicks, hi: LONG CARDINAL],
      hiShort => [lower: MSTicks, higher: CARDINAL],
      ENDCASE
    ];
```

ReadEventTime returns the current time of day, as an **EventTime.**

```
Intime.ReadEventTime: PROCEDURE RETURNS [EventTime];
```

These functions perform arithmetic on **EventTimes**, or answer questions about them. **IsLaterTime** is **TRUE** if **t1** is later than **t2**. The result of **EventTimeDifference** is positive under the same circumstances.

MsTicksToDeltaTicks returns its input, converted to **DeltaTicks** units; the **rem** return value is approximately **rem_s-(dT*msPerDeltaTick)**. This is clearly intended for a very specialized use, to which it is put in IncreekImpl.mesa.

```
Intime.EventTimeDifference: PROCEDURE [t1, t2: LONG POINTER TO READONLY Intime.EventTime]
    RETURNS [Intime.MsTicks];
```

```
Intime.IsLaterTime: PROCEDURE [t1, t2: Intime.EventTime] RETURNS [BOOLEAN] ;

Intime.AddDeltaTimeToEventTime: PROCEDURE [eT: Intime.EventTime, dT:
    Intime.DeltaTime] RETURNS [rT: Intime.EventTime];

Intime.SubtractMsTicksFromEventTime: PROCEDURE [
    eT: LONG POINTER TO Intime.EventTime, ticks: Intime.MsTicks];

Intime.MsTicksToDeltaTime: PROCEDURE [s: Intime.MsTicks]
    RETURNS [dT: Intime.DeltaTicks, rem: Intime.MsTicks];
```

InOS

The **Inscript** facilities have been implemented for both the Alto/Mesa and the Pilot/Mesa versions of Cedar. The source changes have been limited to two implementation modules: the implementation of **ClassInscript**, and the implementation of an interface intended to confine operating system dependent functions, **InOS**.

InOS contains redefinitions of most of the **CWF** formatted-write functions, for two reasons. The first is that different initialization procedures are needed to use these functions in the two OS worlds (the implementation of **InOS** is responsible for providing default initialization if the functions are to be used). The second is that a "production" implementation may choose to supply dummy procedures for these operations -- they are used only for diagnostic purposes in the **Inscript** package.

Procedures

```
InOS.WF0: PROC [s: STRING];
. . .
InOS.WF4: PROC [s: STRING, a,b,c,d: LONG POINTER];
InOS.WF: PROC [s: STRING, a,b,c,d: LONG POINTER _ NIL];
InOS.WFN: PROC [s: STRING, array: DESCRIPTOR FOR ARRAY OF LONG POINTER];
InOS.WFC: PROC [CHARACTER];
InOS.WFCR: PROC;
InOS.SetCode: PROC[CHARACTER, PROC[LONG POINTER, STRING, PROC[CHARACTER]]];
InOS.SetWriteProcedure: PROC [PROC[CHARACTER]] RETURNS [PROC[CHARACTER]];
InOS.WFError: ERROR;
```

The remainder of this interface will be of interest only to those providing new implementations of **ClassInscript** and/or **Interminal**. It supplies procedures for dealing with the keyboard hardware, the real time facilities, and the high-priority process activities required to provide the keyboard "interrupt" process (supplied by **InterminalImpl**.)

These functions merely return the fixed long addresses that point to the states of the keyboard, mouse, and cursor. They appear here because Pilot supplies functions obscuring the absolute locations, while the Alto world assumes that the client knows the addresses.

```
InOS.GetKeyboard: PROC RETURNS[LONG POINTER TO READONLY UNSPECIFIED];

InOS.GetMousePosition: PROC RETURNS[LONG POINTER TO READONLY UNSPECIFIED];

InOS.GetCursorPosition: PROC RETURNS[LONG POINTER TO UNSPECIFIED];
```

Similarly, the following functions return the current values of the high-resolution interval timer (**ReadFastClock**), and the long-term seconds-resolution clock (**GetTimeParameters**). The latter function also returns a value corresponding to the number of interval-timer ticks in a second; in addition, it should perform any initialization required to perform these two functions (an issue in the current Alto implementation).

```
InOS.ReadFastClock: PROC RETURNS [LONG CARDINAL];

InOS.GetTimeParameters: PROC RETURNS [seconds: LONG CARDINAL,
    fastOneSecond: LONG CARDINAL]; -- pulses per second
```

SetupTerminalHandler must perform any OS-dependent activities related to shutting off any existing terminal handlers and starting ours. **WaitForTick** provides a non-busy wait until the next process-scheduling interval (approx. 60 hz.). **MakeCallerResident** must lock the code segment of

the caller into physical memory.

```
InOS.SetupTerminalHandler: PROC;
InOS.WaitForTick: PROC;
InOS.MakeCallerResident: PROC;
```

ClassInscript and Interminal

The **ClassInscript** interface provides an **Inscript** object, whose function is to record variable-length entries in a disk file, and to supply these entries on request at a later time. In addition, the **Inscript** (with the help of its clients) periodically records sufficient information to allow the current output position to be reestablished after a system crash or other uncontrolled termination of the package. Beyond that, the **Inscript** knows little of the semantics of the entries it records. For the courageous implementor, there follows a brief description of the **ClassInscript** interface. Following that is an even briefer exposition of the current implementation.

Data Types

An **Inscript** object represents the inscript file as a whole, along with all the operations pertaining to it. The **InscriptPageDescriptor** describes a particular page within the file, and a position within that page; there may be more than one of them. It does not supply any operations. In retrospect, probably the **InscriptPageDescriptor** should become the **Inscript** object. This would reduce considerably the clumsiness of some of the following specifications.

There are two kinds of **Inscript** clients; those who wish to record into the inscript -- *recording clients* -- and those who wish to read entries from them -- plain old clients. At present there is but one recording client for an inscript, although that is not a requirement. Recording clients will typically wish to read as well.

An **InscriptPageNumber** is a "virtual page number" of a page in the **Inscript** file.

```
ClassInscript.Inscript: TYPE = REF InscriptObject;
ClassInscript.InscriptPageDescriptor: TYPE = REF InscriptPageDescBody;
ClassInscript.InscriptPageNumber: TYPE = INTEGER;
```

During initialization, the recording client must supply a **COmProc** and a **KEyProc** to assist in reestablishing the current "end of file" position within the inscript file. **KEyProc** should assume that **descriptor** is positioned at the beginning of a page, and should extract an **OrderKey** from that page (this clearly requires that the client must record an **OrderKey** at a known place within each page as it writes the page.) **COmProc**, given two **OrderKeys**, should return TRUE iff **a** is "less than" **b**. In the **Increek** implementations, **OrderKeys** are **EventTimes**. The **Increek** implementation knows nothing about **OrderKeys** except their size.

```
ClassInscript.OrderKey: TYPE = RECORD [a, b, c: WORD];
ClassInscript.COmProc: TYPE = PROCEDURE [a, b: ClassInscript.OrderKey] RETURNS [aLessB:
    BOOLEAN];
ClassInscript.KEyProc: TYPE = PROCEDURE [
    inscript: ClassInscript.Inscript, descriptor: ClassInscript.InscriptPageDescriptor]
    RETURNS [ClassInscript.OrderKey];
```

In order to read entries from the inscript, the client must supply a **ReadAssignment** procedure. Its job is to coerce the LONG POINTER argument, assumed to denote an entry of interest to the client, into a comprehensible data type, determine the entry's length, copy the entry to a new location if desired, and return the length of the entry to the caller, for use in advancing to the next entry..

```
ClassInscript.ReadAssignment: TYPE = PROCEDURE [p: LONG POINTER TO UNSPECIFIED]
    RETURNS [wordsToAdvance: CARDINAL];
```

WaitMode is used in the primitive **WaitForEntry** function used to implement the waiting activities in **ClassIncreek.GetAction**. Its use is described in detail in the **Increek** section.

```
ClassInscript.WaitMode: TYPE = {forever, dontWait, timed};
```

```
ClassInscript.InscriptErrorCode: TYPE = {
    entryOutOfBounds, -- trying to position out of bounds or old stuff has disappeared
    invalidInscriptSpecs, -- bad arguments to NewStd...
    invalidInscriptFile, -- while opening old inscript file
    descriptorUninitialized -- in AdvancePage--,
    invalidPageKey -- in KeyProc during intitialization --};
```

Procedures

NewStdInscript creates an **Inscript** using the standard implementation, discussed below. In addition to the obvious parameters, **initializeFile** will force a reinitialization of the inscript file even if it already exists, the inscript file will contain 2^**lnFileSize** data pages, and the inscript history will be discarded in groups of 2^**lnGroupSize** pages to make room for new entries. The **Release** operation cleanly terminates an input session (no one calls this one, in current implementations.) **GetPageLimits** returns the "virtual page numbers" of the pages that currently exist within the inscript file. Functions like **ClassIncreek.SetAtTime** use this information to initialize their search activities.

```
ClassInscript.NewStdInscript: PROCEDURE [fileName: STRING,
    KeyProc: ClassInscript.KEyProc, ClassInscript.ComProc: COmProc, initializeFile:
    BOOLEAN _ FALSE,
    lnFileSize: CARDINAL_7, lnGroupSize: CARDINAL_4]
    RETURNS [ClassInscript.Inscript];

ClassInscript.Release: PROCEDURE [self: ClassInscript.Inscript] RETURNS [nilInscript:
    ClassInscript.Inscript];

ClassInscript.GetPageLimits: PROCEDURE [self: ClassInscript.Inscript]
    RETURNS [earliestPage: ClassInscript.InscriptPageNumber, latestPage:
    ClassInscript.InscriptPageNumber] ;
```

An **InscriptPageDescriptor**, when created, does not refer to any inscript page; other operations make it valid. **CopyPageDescriptor** copies the information of one descriptor into another, maintaining any necessary internal reference counts (don't perform this function any other way.) **ReleasePageDescriptor** discards one.

```
ClassInscript.CreatePageDescriptor: PROCEDURE [self: ClassInscript.Inscript]
    RETURNS [rDescriptor: ClassInscript.InscriptPageDescriptor];

ClassInscript.CopyPageDescriptor: PROCEDURE [self: ClassInscript.Inscript, dest:
    InscriptPageDescriptor,
    source: ClassInscript.InscriptPageDescriptor];

ClassInscript.ReleasePageDescriptor: PROCEDURE [self: Inscript, descriptor:
    InscriptPageDescriptor]
    RETURNS [nilPageDescriptor: InscriptPageDescriptor];
```

Using the **KEyProc** and the **COmProc**, the **ClassInscript** implementation must establish a valid output position when the **Inscript** object is created. Subsequently, the only valid recording operations are **SetWritePage** and **WriteEntry**.

WriteEntry records the value of the entry array into the next available locations in the current page (using internally maintained page descriptors), failing if there is not enough room. The recording client must then call **SetWritePage** in order to move on. Things are done this way so that the client can record per-page information (e.g., the page's **OrderKey**) at the beginning of each page. After initialization, the client should call **WriteEntry** without any initial call to **SetWritePage**, and the initialization must take care of guaranteeing that this works -- the output position must be set at "end of file".

SetWritePage will advance to the next "virtual" page, increasing the value of **GetPageLimits[...].latestPage** by one, and possibly increasing **ditto.earliestPage**, thus destroying some history. It will fail only under the circumstances (which should be made vanishingly

unlikely) that no physical page in the inscript file is available for use. On return, the page descriptor will represent the first data position within the new page.

```
ClassInscript.WriteEntry: PROCEDURE [self: ClassInscript.Inscript, entry: LONG DESCRIPTOR FOR
    ARRAY OF WORD]
    RETURNS [success: BOOLEAN];

ClassInscript.SetWritePage: PROCEDURE [self: ClassInscript.Inscript] RETURNS [success:
    BOOLEAN];
```

To read from the inscript, a client performs **ReadEntry** operations, supplying a **ReadAssignment** procedure in order to interpret the entries and determine their length. When **ReadEntry** fails, no more entries remain in the current page; the client should next call **SetPage** or **AdvancePage**. Again, this is done to allow the client to do something special at page boundaries. If the **InscriptPageDescriptor's** page disappears to make way for new entries, **ReadEntry** will raise **InscriptError[code: entryOutOfBounds]**.

SetPage adjusts its **InscriptPageDescriptor** to the beginning of the indicated "virtual" page, failing if that page no longer exists or doesn't exist yet. **AdvancePage** performs **SetPage** of "current plus one." If **AdvancePage** fails, it is because no further entries exist in the file. The client might then call **WaitForEntry**, which will return only if a timeout condition (described in the Increek section) is satisfied, or if one or more new entries have been added to the inscript since the operation was invoked.

```
ClassInscript.ReadEntry: PROCEDURE [self: ClassInscript.Inscript, descriptor:
    ClassInscript.InscriptPageDescriptor, Read: ClassInscript.ReadAssignment]
    RETURNS [success: BOOLEAN];

ClassInscript.SetPage: PROCEDURE [self: ClassInscript.Inscript, descriptor:
    ClassInscript.InscriptPageDescriptor,
    pageNumber: ClassInscript.InscriptPageNumber] RETURNS [success: BOOLEAN];

ClassInscript.AdvancePage: PROCEDURE [self: ClassInscript.Inscript, descriptor:
    ClassInscript.InscriptPageDescriptor]
    RETURNS [success: BOOLEAN];

ClassInscript.WaitForEntry: PROCEDURE [self: ClassInscript.Inscript, waitMode:
    ClassInscript.WaitMode, waitInterval: Intime.MsTicks _ 1000,
    waitStartTime: LONG POINTER TO Intime.EventTime _ NIL]
    RETURNS [moreEntries: BOOLEAN];

ClassInscript.InscriptError: ERROR [code: ClassInscript.InscriptErrorCode];
```

In the existing package, **InscriptImplAlto** and **InscriptImplPilot** provide two functionally equivalent implementations of **ClassInscript**. **InterminalImpl** acts as the recording client, polling the terminal hardware once a tick and recording any changes it sees. This implementation records its **Actions** into a disk file that behaves as a large ring buffer, filling seldom more than one Alto-sized page per minute under normal conditions. Thus **Actions** are remembered for quite some time, but not indefinitely (the standard **Increek** uses a 128 page file.) Both implementations run the recording client at high priority. They both lock inscript pages into memory to prevent page fault delays. They seem to perform efficiently, without losing events. They are monitored so that clients from more than one process can access their inscripts.

The interfaces were designed so that other implementations of **Inscripts** could be provided, for instance to immutably record user **Actions**, possibly condensed (retaining fewer actions), to supply an application with **Actions** from a source other than the attached terminal, etc. These possibilities are discussed in more detail in the design document cited as a reference.