Plexil Reference - plexil

About this chapter
This chapter documents the PLEXIL programming language, and how to create and compile Plexil files for execution.

We explain PLEXIL primarily though examples, using PLEXIL's standard programming syntax, simply called Plexil, or sometimes *standard Plexil* to distinguish it from other represesentations of PLEXIL such as Plexilisp or Plexil's XML form.

This is not a rigorous treatment of PLEXIL, but hopefully -- in combination with the Example PLEXIL Plans -- provides enough to enable you to program in PLEXIL effectively.

The PLEXIL langauge is summarized briefly in the previous chapter, Overview. Its execution semantics are explained more deeply in the following chapter, Detailed Semantics, which cites papers that provide a rigorous formal definition of the language. The PLEXIL syntax is formalized in XML schemas found in the software distribution.

NOTE: Although this chapter focuses on the Plexil language and its semantics, as independent from the exact behavior of a particular implementation of Plexil, explanations of how the supplied Plexil Executive behaves have been provided in places we felt this information helpful.

Actions
A PLEXIL plan consists of one *action*, which may have subactions. An action is a structure specifying a certain kind of behavior.

An action is notated by a pair of curly braces containing the action's specification, preceded optionally by a name for the action. The simplest action is the empty action (also called an *empty node*):

```
{ }
```

We can give it a name:

```
DoNothing : { }
```

The action's name (also called its *Id*), denoted by an identifier and colon preceding the opening brace, is optional. An anonymous (nameless) action is valid, though it cannot be referenced anywhere explicitly, except within the action itself (by using the `Self` keyword, described below). In practice, every action has a name; an anonymous action is assigned a unique name when compiled into PLEXIL's XML form for execution.

An action and its parent, immediate children, and siblings (this structure will be explained later) must have unique names. Uniqueness of names across more distant relationships in a plan is not required, especially since these actions cannot reference each other (more on referencing scope below).

Plexil is case-sensitive, but whitespace-insensitive, so the `DoNothing` plan above can also be written, for example, in either of the following ways:

```
DoNothing:
{
}
```

```
DoNothing: {
}
```

Through composition, actions form a tree-shaped hierarchy. The root of the tree is the *root* or *top level* action. A Plexil file must contain exactly one top level action.

Actions have two components: a set of *attributes* that drive the execution of the action, and a "body" which specifies what the execution of the action accomplishes.

Actions which have no attributes may omit the enclosing braces. Examples will be provided below.

Action Attributes

Actions may contain *attributes*, which include local variables, an *interface*, *conditions*, and a comment. Attributes are optional, and some have specific default values. When attributes are specified, they must occur *first* in action's form, i.e. immediately following the opening curly brace.

Variables

An action may declare variables, which are local to the action. Plexil currently supports variables of type Boolean, integer, real, string, and arrays of these four basic types. Examples of declarations of the basic types are as follows.

```
Boolean isReset;
Integer n;
Real pi;
String message;
```

These examples of variable declarations do not specify initial values for the variables. Uninitialized variables of all types except arrays are given the value Unknown. Here are the same variable declarations with initial values specified. Initial values must be literals -- expressions are not allowed.

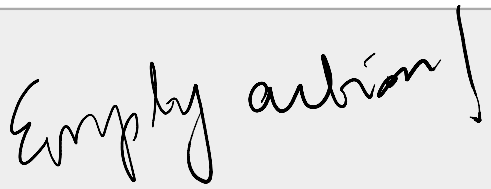
```
Boolean isReset = true;
Integer n = 123;
Real pi = 3.14159;
String message = "hello there";
```

Arrays are declared by following the variable name with square brackets containing the size of the array. Array variables do not default to Unknown, but rather to an allocated array, all of whose *elements* are Unknown. The first example below declares an array of 100 integers. The second declares a smaller array of real numbers, with the first three elements initialized (the remaining seven are Unknown).

```
Integer scores[100];
Real defaults[10] = #(1.3 2.0 3.5);
```

Variables have *lexical scope*, which mean they are visible only within the action and any descendants of the action. Scope can be explicitly limited using the Interface clause described below. Here is an example of an empty action that declares some variables.

```
DoNothing1:
{
  String name = "Fred";
  Real MaxTemp = 100.0;
}
```



So far we've using empty actions as examples simply because we haven't yet introduced the other actions. The example above is illustrative but would serve no practical purpose, since its variables cannot be used in any way.

NOTE: Variable declarations and interface declarations (described in the following section) must occur prior to any other kinds of attributes in an action definition. They may be intermixed.

Interface

An action's *interface* is the set of variables it can read and/or write (assign) to. By default, the interface of an action N is the union of its parent's interface and the variables declared in N. Interface clauses impose a *restriction* on the set of variables inherited from the parent's interface by specifying the *only* variables from the parent that are accessible. There are two kinds of interface clauses. The `In` clause specifies variables that can be read. The `InOut` clause specifies those that can be read or written. All stated variables must be part of the parent's action's interface, otherwise the clause is in error. Furthermore, read-only variables in the parent cannot be declared `InOut`. Here's an example of an empty action with interface clauses.

```
Test:
{
  In Integer x, y;
  InOut String z;

  Integer a, b;
}
```

Variables x and y are assumed to be readable, and z readable and writable, in Test's parent action. No other variables in Test's ancestors will be accessible. Variables a and b are local variables in Test.

An action's interface variables are also called its *parameters*. It is an error for an action to declare a variable having the same name as a variable that appears in its interface.

NOTE: Variable declarations (described in the previous section) and interface declarations must occur prior to any other kinds of attributes in an action definition. They can be intermixed.

Conditions

An action can specify up to seven *conditions* that govern precisely how the action is executed.

```
 StartCondition <logical expression>
 EndCondition <logical expression>
 RepeatCondition <logical expression>
 SkipCondition <logical expression>
 PreCondition <logical expression>
 PostCondition <logical expression>
 InvariantCondition <logical expression>
```

A condition specifies a Plexil *logical expression*. Expressions are described in a section below. Here are some varied examples of conditions:

```
StartCondition Action1.outcome == SUCCESS;

EndCondition SignalEndOfPlan.state == FINISHED ||
             SendAbortUpdate.state == FINISHED ||
             abort_due_to_exception;

PreCondition Request_Human_Consent.state == FINISHED &&
             Lookup(ZZZZCWEC5520J) == 1;

PostCondition AtGoal;
```

```
InvariantCondition Lookup(ZZZZCWEC5520J) == 1;

RepeatCondition Count < 10;
```

Here is an example of an empty action with some declarations and conditions:

```
Step2:
{
 Real temperature;
 Real MaxTemp = 100.0;

  StartCondition Step1.state == FINISHED;
  InvariantCondition temperature < MaxTemp;
}
```

The conditions specify that this action should begin execution after action Step1 finishes, and that the temperature should remain less than MaxTemp throughout execution. (Note that Plexil does not provide *constants*, only variables). Incidentally, this an an example of a potentially useful empty action. Empty actions are often used to *wait* for a condition (expressed through the start condition) and/or to test or *verify* a condition (expressed here through the invariant condition).

There are two kinds of comments in a Plexil plan. The source code can include comments to help document the code but that are not preserved in the translated Core PLEXIL XML output. These are notated in the C/C++ style syntax for block and single line comments. Examples of each are as follows.

```
/*
 * Here is a block comment example which
 * allows for multiple lines.
 */

// Here is a single-line comment example that extends to the end of the line.
```

Second, Plexil actions have the option of including a single `Comment` clause, which must be the first item in the action's attribute section. Here's an example:

```
Comment "This action verifies the robot's camera is functioning.";
```

The `Comment` clause gets preserved in the compiled (XML) version of the plan, unlike other comments.

Leaf Actions

As described in the Overview, PLEXIL has many kinds of actions. The type of a given action is identified by the action's *body*. An action's body is what immediately follows its attributes (described in the previous sections).

Actions that do not contain or decompose into subactions form the leaves in a PLEXIL plan tree. These actions are called *nodes* and are part of Core PLEXIL, which is the subset of PLEXIL that is executed directly.

Empty Action

All the examples presented above are empty actions (also called empty nodes). Empty actions contain only attributes. They have no external behavior (i.e. no direct effect on an external system or a plan variable). In practice, empty actions are quite useful and common. A typical use is for verification of a state in the external world. Here's an action that verifies a temperature reading.

```
VerifyTemp:
{
  PostCondition Lookup(engine_temperature) > 100.0;
}
```

Assignment

An assignment to a declared variable has the following form:

```
<variable> = <expression>;
```

The `<variable>` part of the assignment, referred to as its left-hand side (LHS), must be a writable variable in the action's interface. The `expression`, referred to as the right-hand side (RHS) of the assignment, can be any PLEXIL expression. Expressions are described below. The type of the expression must match the type of the variable.

An *assignment action*, or *assignment node*, is one that contains only an expression of this form in its body. The following are examples of assignment actions. Note that some context, in particular the variables' declarations, are not shown.

```
IncrementCounter:
{
  ExecutionCount = 1 + ExecutionCount;
}

CopyEntry:
{
  TemperatureReadings[i] = x;
```

```
}
```

Assignment actions without attributes may omit the braces. The preceding examples could be rewritten as:

```
IncrementCounter: ExecutionCount = 1 + ExecutionCount;

CopyEntry: TemperatureReadings[i] = x;
```

Assignments and Concurrency

If two nodes in a PLEXIL plan attempt to assign the same variable simultaneously, this is an error condition. The PLEXIL compiler does not detect the possibility of concurrent assignment, and unfortunately the current PLEXIL executive behaves ungracefully when it is attempted: it issues a message about the conflict and then aborts. If your plan contains such nodes, this contention problem can be resolved with the `Priority` clause. Here's a trivial contrived example:

```
ConcurrentAssignment: Concurrence
{
  Integer x;

 A:
  {
    Priority 1;
    x = 0;
  }

 B:
  {
    Priority 2;
    x = 1;
  }
}
```

Without the `Priority` clauses, a runtime error would result. The `Priority` clause *orders* the execution of nodes from the lowest priority number to the highest. In this example, node A will execute first, then B, and the final value of x will be 1. Though it's possible that the `Priority` clause may have a legitimate application, it is probably best to design your plans such that potentially conflicting assignments are avoided.

Command

A *command* has the form:

```
[<variable> =] <command_name> ([<argument_list>]);
```

where:

- `command_name` is an identifier or a parenthesized string expression;
- `argument_list` is an optional comma-separated list of zero or more arguments, which may be either literal values, variables, or array element references (other kinds of expressions are not supported).

The assignment of the command's return value (assuming it returns a value) to a variable is optional, and if specified, `<variable>` must be a writable variable in the action's interface.

A *command action*, or *command node*, is one that contains only an expression of this form in its body. The following are examples of command actions. Note that some context is not shown, e.g. the declaration of the command (discussed next) and that of the variable receiving the return value.

```
StopRover: { stop(); }

SetWaypoint: { set_waypoint (x,y,z); }

GetSpeed: { speed = get_speed(); }
```

As with assignment actions, if no attributes are required, the braces may be omitted:

```
StopRover: stop();

SetWaypoint: set_waypoint (x,y,z);

GetSpeed: speed = get_speed();
```

Commands must be declared at the top of the file in which they are used. Here are some example declarations:

```
// simple command
Command stop();

// command with parameter (parameter name is optional)
Command Drive(Real meters);

// command with return value
Boolean Command TakePicture(Integer, Integer, Real);
```

Asynchronous command execution

Calls to commands do *not* wait, or block execution of the plan. By default, command nodes finish when the executive receives acknowledgement (via the command's *handle*) of the command from the external interface, which occurs almost immediately.

The command executes in the external system asynchronously with the plan. To examine the progress of the command, the plan should inspect its handle. See the description of `command_handle` below); for a deeper discussion, see Resource Model.

Note that a command may take arbitrarily long to complete in the external system. If the command returns a value, and this value is assigned to a variable in the command action, the action should *wait* for the value, i.e. the command's completion, before ending execution. This is accomplished with an appropriate end condition. Here's an example:

```
ConfirmProceed:
{
  Boolean result;
  EndCondition isKnown(result);
  PostCondition result;
  result = QueryYesNo("Proceed with instructions?");
}
```

In this example, the end condition makes the action wait for the command's result (whose initial value is Unknown). It further stipulates, via the postcondition, that *success* of this action requires a positive user confirmation.

However, this idiom is cumbersome to code and difficult or impossible to get right in the general case. For example, if a command assigns to a variable that already has a value, the `isKnown` test is unhelpful. Fortunately, Plexil provides a convenient form for *synchronous* commanding -- see the section below.

Optionally, command nodes may specify resource requirements for the affected command. The syntax and semantics for this is described in the Resource Model chapter.

Utility Commands

Several convenient utilities, in the form of commands, are available in Plexil. Currently there are two commands that print PLEXIL expressions to the standard output stream (e.g. the Unix terminal).

- `print (exp1, exp2, ...)` prints expressions without any added characters.
- `pprint (exp1, exp2, ...)`, short for "pretty print" is like `print` but adds spaces between the expressions and a final newline.

The utility commands are automatically available when running Plexil through the Test Executive.

Otherwise they are available by including the Utility Adapter.

Update

An Update action/node serves to relay information outside the executive. For example, it can be used to update a planner or other system that has invoked the executive, with status about execution of the plan. The manner in which this information is sent is determined by the external interface for the executive. An update consists of name/value pairs; an update should include one or more such pairs. The **`Update`** keyword identifies an Update node, and has the form:

```
Update <name> = (<value> | <variable>) [, <name> = (<value> | <variable>) ]*;
```

where `name` is an identifier, and the right hand side is either a `value` which is a literal (e.g. 5, "foo"), or a `variable` which is an identifier naming a declared variable that is visible to the action. Any number of such name/value pairs can be given, separated by commas.

Here's an example:

```
SendAbortUpdate:
{
  StartCondition MonitorAbortSignal.state == FINISHED;
  Update taskId = taskTypeAndId[1], result = -2, message = "abort";
}
```

As with assignment and command actions, if no attributes are required, the containing braces may be omitted.

Library Call

A `Library Call` action/node has the following form as its body.

```
LibraryCall <Id> [<alias_list>];
```

where `<Id>` is the ID of the invoked *library action* (or *library node*). The `<alias_list>` is an optional list of *aliases*, which are pairs of the form

```
<parameter> = <expression>
```

An alias allows one to rename/assign an action parameter (i.e. a variable present in the interface of the library action) with an actual value or declared variable.

Here's a contrived example of a call to trivial library action. The first file defines the library action `F`, and the second file contains an action that calls `F`.

```
--- begin F.ple ---

F:
{
 In Integer i;
 InOut Integer j;
 j = j * j + i;
}

---- end F.ple ----

--- begin LibraryCallTest.ple ---

LibraryAction F (In Integer i, InOut Integer j);

LibraryCallTest:
{
 Integer k = 2;
 LibraryCall F(i=12, j=k);
}
```

A library action can be any type of action (e.g. Sequence, Command) but it must be a *top level* action, that is, the outermost action in a file. Conversely, any top level action can be used as a library action.

As with assignment, command, and update actions, if no attributes are required, the containing braces may be omitted:

```
LibraryAction makePhoneCall(In Integer number);

CallHome: LibraryCall makePhoneCall(number=5551212);
```

Library Call execution

Prior to execution of a Plexil plan, at every point of a library call, a copy of the invoked library action is *statically* inserted in place of the call. Hence, "call" is technically a misnomer, and the mechanism for library execution is essentially a "macro" style code substitution. The executed plan is a single monolithic action with all library calls replaced by their invoked actions. Thus, repeated "calls" to the same library action can produce a large plan for execution.

High-level Actions

Actions that decompose into sub-actions are described in this section. These actions are translated into Core PLEXIL nodes prior to execution. Core PLEXIL consists of the leaf actions (nodes) described in the

previous section, plus the List Node, described in this section under Concurrence.

Sequence

A Sequence executes its sub-actions in the given order. If any action fails (i.e. terminates with status FAILURE), the Sequence also terminates with status FAILURE. A Sequence succeeds if and only if all its actions succeed. An empty Sequence always succeeds.

A Sequence is denoted as follows.

```
Sequence
{
  <action1>;
  ...
  <actionN>;
}
```

Because the Sequence is, in practice, the most common construct in Plexil plans, the `Sequence` keyword is optional:

```
{
  <action1>;
  ...
  <actionN>;
}
```

Unchecked Sequence

An Unchecked Sequence is just like a Sequence, except that the outcome of each sub-action is not checked. All actions will be executed, in the given order, and an unchecked sequence succeeds by default.

```
UncheckedSequence
{
  <action1>;
  ...
  <actionN>;
}
```

Concurrence

The **`Concurrence`** keyword identifies a Concurrence action and encloses zero or more *child* actions, which are executed *concurrently*. Precisely, there are no execution constraints on the sub-actions other than those imposed by explicit conditions (those found in each sub-action as well as in the Concurrence form

itself).

```
Concurrence
{
  <action1>;
  ...
  <actionN>;
}
```

A Concurrence finishes when all its children have finished. If a different behavior is desired, such as ordering constraints between children, or finishing before all children have executed, this behavior must be specified through explicit conditions in the children. Here is a contrived example that illustrates a Concurrence with a particular execution protocol:

```
Command inform(String message);
Boolean Command DoIt(Integer n);

Root: Concurrence
{
  Integer x;

  Inform:
   inform("Plan executing...");

  Init:
    x = GetX();

  Commence:
  {
    Boolean result;
    StartCondition Init.state == FINISHED;
    PostCondition result;
    SynchronousCommand result = DoIt(x);
  }

  InformSuccess:
  {
    StartCondition Commence.outcome == SUCCESS;
    inform("Operation succeeded!");
  }
```

```
  InformFailure:
  {
    StartCondition Commence.outcome == FAILURE;
    inform("Operation failed!");
  }
}
```

In the example above, the Inform and Init actions are unconstrained -- they can start immediately. The Commence action waits for Init to finish before it can start. After it finishes, either InformSuccess or InformFailure will execute, depending on the result.

NOTE: If more than one child action is eligible for execution at a given moment, and PLEXIL is being executed on a sequential machine, the actual order of execution is *unspecified*. In any context where the exact execution order of actions really matters, it must be encoded explicitly in the plan.

Try

In this kind of sequence, the sub-actions are executed in sequence, *until* one succeeds. The remaining actions are skipped. A Try succeeds if and only if one of its actions succeed. An empty Try always fails.

```
Try
{
  <action1>;
  ...
  <actionN>;
}
```

If-Then-Else

This is a variant of the traditional *if-then-else* construct, with optional "elseif" and "else" parts. The `if` and optional `elseif` clauses each specify a condition, and an action to execute if this condition is true; they are evaluated in the order listed until one condition succeeds. The optional `else` clause provides a default action which is executed if none of the conditions evaluates to true. The `if` action must be terminated by the `endif` keyword.

Each clause may have multiple child actions.

```
if C1 then
  <action-1>
[elseif C2
  <action-2> ]*
[else
  <action-3> ]
```

```
endif
```

where C1, C2 are logical expressions. Specifically, if C1 evaluates true, action-1 will be executed. If C1 is false or *unknown*, C2 is then evaluated, etc. If an `if` statement has no true conditions, and does not supply an `else` clause, it will invoke no action.

Examples:

```
if true
  {
    foo();
    bar();
  }
elseif 2 == 2
  bar();
else
  baz();
endif

if ( Lookup(raining) )
  Concurrence
  {
    Wipers: turn_on_wipers();
    Lights: turn_on_lights();
  }
endif
```

While Loop

This is a traditional *while* loop.

```
while C
  <action>
```

where C is a logical expression. Example:

```
while ! Lookup(Rover.WheelStuck)
   Rover.driveOneMeter();
```

For Loop

This is a traditional For loop, limited to a numeric variant.

```
for (T V = Z; C; E)
  <action>
```

where T is either `Integer` or `Real`, V is a variable name, Z is a numeric expression for the initial value of V, C is a logical expression indicating when to continue the loop, and E is a numeric expression for updating V after each iteration. Examples:

```
for (Integer i = 0; i <= 5; i + 1) pprint ("i: ", i);

for (Integer i = 2; i <= n; i + 1)
{
  result = s1 + s2;
  s1 = s2;
  s2 = result;
}
```

OnCommand

`OnCommand` is a "handler" for an external command. It is used in multiple executive settings where one executive receives a command sent by another executive. It has the following syntax.

```
OnCommand <command-name> [<parameter-declaration>]
  <action>
```

where `<command name>` is either a string matching the command name that was executed, or a string variable containing this name; <parameter-declaration> is a list of zero or more comma-separated variable declarations for parameters; and <action> is an action to be performed upon receiving the command. Here's an example:

```
OnCommand "Sum"
  Increment: { SendReturnValues(3); }
```

Every `OnCommand` action is required to call the command

```
SendReturnValues(<value>)
```

If a call to `SendReturnValues` is not present, `SendReturnValues(true)` is called automatically after the sub-action has finished executing. For more information, see Inter-Executive Communication.

OnMessage

`OnMessage` is similar to `OnCommand`, but only receives text sent by the command `SendMessage`, and may

not have parameters.

```
OnMessage <message>
  <action>
```

Example:

```
OnMessage “ConnectionEstablished”
  BeginProcess();
```

This "handler" for messages is invoked by the following command:

```
SendMessage(<string>)
```

For more information, see Inter-Executive Communication.

Synchronous Command

Plexil provides a convenient form for synchronous commands, which automatically waits for a return value if expected, otherwise a command handle indicating completion of the command.

Furthermore a *timeout* can be given, which specifies the maximum amount of time (expressed as a unitless real number) to wait for the command to complete. When timeout is given, a real-valued *tolerance* can also be given; this specifies a minimum granularity for the time measurement.

Here are some examples. First, we call the command `foo` with no arguments:

```
SynchronousCommand foo();
```

Call `foo` with argument 1 and a timeout of 2.0 time units:

```
SynchronousCommand foo(1) Timeout 2.0;
```

Call `foo`, assigning its return value to `x`.

```
SynchronousCommand x = foo();
```

Call `foo`, assigning its return value to `x`, with a timeout of 2.0 with tolerance of 0.1 (i.e. check the time every 0.1 units if possible).

```
SynchronousCommand x = foo() Timeout 2.0, 0.1;
```

This section begs elaboration of several aspects of PLEXIL not yet discussed in detail.

- Time. As mentioned in the Overview, time is not a special concept in PLEXIL -- it's just an external world state; specifically, a real-valued state variable named `time`. This variable may be referenced explicitly, e.g. `Lookup (time, 1)`, though in most cases it is used implicitly: the Plexil executive reads it from the external world at every cycle and uses it for time-related computations in a plan, such as the timeout in `SynchronousCommand` described here. The tolerance parameter to the timeout is simply the tolerance given to the Lookup that queries `time` for this action.

- Command Handles. These are described in the Resource Model chapter, but we must note here that instances of SynchronousCommand without return values *require* that certain command handles are supported by the Plexil application. Specifically, for SynchronousCommand to work, the application *must* return one of the following handles for the command invoked: COMMAND_SUCCESS, COMMAND_FAILED, COMMAND_DENIED.

Wait

The `Wait` action does just that -- waits for a specified amount of time to pass:

```
Wait <time-units> [<tolerance>]
```

where `<time-units>` is a unitless real number (the time unit this actually represents is application-specific), and so is `tolerance`. Tolerance, which is optional and defaults to the <time-units>, specifies the minimum amount of time that is of significance in the wait. Real-valued variables can also be used. Examples:

```
Real rtol = 0.5;
Real rdelay = 1.414;
```

```
Wait 2.0;           // wait 2.0 units
Wait 5.0, 0.1;      // wait 5.0 units with a tolerance of 0.1 units
Wait rdelay, rtol;  // wait, using variables
Delay1: Wait 3.10;  // a wait action named Delay1
```

Data Types and Expressions

PLEXIL supports the following data types: integer, real, string, Boolean (logical expressions), and arrays (homogeneous arrays of any type except array itself). PLEXIL provides a variety of operations on each of these types.

An *expression* in PLEXIL is either a literal value, a variable, a lookup, or a combination of any of these formed by operators. In particular, expressions can contain expressions (i.e. they can be arbitrarily

complex). Expressions can occur within action conditions, the target of assignments, and resource specifications.

Unknown/isKnown

Each PLEXIL type is extended by a special value `UNKNOWN`, i.e. any expression can evaluate to `UNKNOWN`. The unknown value occurs in the following cases.

- It's the default initial value for variables, an action's outcome, and array elements.
- It results when a lookup fails.
- It results when a requested *node timepoint* has not occurred. Node timepoints are discussed below.
- It is a valid value for Plexil logical expressions.

The `UNKNOWN` value is *not* a literal -- it may not be used in a PLEXIL plan. It is tested solely through the `isKnown` operator, which returns false if its argument evaluates to `UNKNOWN`, and true otherwise. An example of the use of `isKnown` is found in the section above on Command actions.

Numeric Expressions

Numeric expressions include literals (integers, real numbers), variables (of type Integer or Real), lookups and node timepoint values (both discussed below), and arithmetic operations: addition, subtraction, multiplication, division, square root, minimum, maximum, and absolute value. In addition, arrays have as numeric expressions their size, element index, and, for arrays of numeric type, their elements.

Here are varied examples of each of the aforementioned types of numeric expressions.

```
234
12.9
X /* where X was declared Integer */
Bar /* where Bar was declared Real */
Lookup(ExternalTemperature)
TakePicture.EXECUTING.START  /* a node timepoint */
Bar + 4.5
X - (30 + Lookup(x) )
3 * X
(3 * X)/(X - 20)
sqrt(X)
abs(X)
Entries[X] /* where Entries is an array of Integer or Real */
```

Precedence and associativity rules for these operators are consistent with the standard rules for C and C++. Parentheses can be used to make explicit the intended semantics.

Integers and Reals may be mixed in Real-valued numeric expressions. Integer values are automatically

promoted to Real in mixed calculations, so are legal in all contexts where a Real is expected. However, a Real value cannot be used where an Integer is expected, e.g. as an array index, nor can a Real value be assigned to an Integer variable or array element.

Numeric Type Conversions

Plexil offers the following type conversion operators for converting a Real to an Integer:

```
ceil(r) /* returns least positive integer greater than or equal to r */
floor(r) /* returns most positive integer less than or equal to r */
round(r) /* as defined in the C language standard */
trunc(r) /* rounds toward 0 */
real_to_int(r) /* For converting a Real that is known to be exactly integer-valued
*/
```

In each conversion function, if the supplied Real is out of range for an Integer, UNKNOWN is returned. Additionally, `real_to_int` will return UNKNOWN if the supplied Real is not exactly an integer value.

Logical Expressions

PLEXIL employs a *ternary* logic, extending the usual Boolean logic with a third value, Unknown, described in a section above. Though strictly a misnomer, the term Boolean is used throughout this manual and PLEXIL itself to describe operators, expressions, and values in this ternary logic.

Logical expressions include the Boolean literals `true` and `false`, `Boolean`-typed variables, lookups, comparisons, logical operations, array elements (of `Boolean` arrays), and the `isKnown` operator.

The logical connectives, their syntax in PLEXIL, and arity (number of operands allowed) are as follows:

```
Negation (Not)     !, NOT      1
Conjunction (And)  &&, AND     2 or more
Disjunction (Or)   ||, OR      2 or more
Exclusive Or       XOR         2
```

When restricted to Boolean (`true` or `false`) values in their constituents, logical expressions in PLEXIL follow the standard rules of Boolean logic. Here is how PLEXIL handles the Unknown value, again a standard interpretation.

```
true && Unknown     = Unknown
false && Unknown    = false
Unknown && Unknown  = Unknown
true || Unknown     = true
false || Unknown    = Unknown
```

```
Unknown || Unknown  = Unknown
true XOR Unknown    = Unknown
false XOR Unknown   = Unknown
Unknown XOR Unknown = Unknown
! Unknown           = Unknown
```

The operators `AND` and `OR` are evaluated left to right in a *short-circuit* fashion. Conjunctions have value `true` until an operand evaluates to `false` or Unknown; this value becomes the value of the expression. Similarly, disjunctions have value `false` until an operand evaluates to `true` or Unknown.

The comparison operators, all of which take exactly two operands, are:

```
Equality                ==
Inequality              !=
Less than               <
Greater than            >
Less than or equal      <=
Greater than or equal   >=
```

In these comparision expressions, if *any* operand evaluates to Unknown, the entire expression yields Unknown.

Here are varied examples of logical expressions.

```
true
false
CommandReceived /* where CommandReceived was declared Boolean */
! CommandReceived
Lookup(RoverInitialized) /* where RoverInitialized is declared a Boolean lookup */
count <= 30 /* where count was declared Integer */
Lookup(RoverBatteryCharge) > 120.0 /* where RoverBatteryCharged is declared a Real
lookup */
Lookup(RoverInitialized) || CommandReceived
Flags[3] /* where Flags is an array of Boolean */
isKnown(val)  /* where val is any variable */
action3.state == FINISHED && action3.outcome == SUCCESS
```

NOTE: Precedence and associativity rules for these operators are consistent with the standard rules for C and C++. Parentheses can be used to make explicit the intended semantics.

String Expressions

String expressions include literal strings, variables (of type String), lookups, and string concatenations. Examples of each are as follows.

```
"foo"
"Would you like to continue?"
Username /* where Username was declared string */
Lookup(username)
"Hello, " + "Fred"    => "Hello, Fred"
"Hello, " + Username
```

The only comparison operations currently defined on strings are == and !=.

The `strlen` operator returns the length of a String as an Integer.

Dates and Durations

One may want to reason about time. PLEXIL provides basic support for *date*, *time*, and *duration* types as defined by the ISO-8601 standard. See http://en.wikipedia.org/wiki/ISO_8601 for a detailed description of this standard and the date/duration formats, as these are covered only by example here.

Date and Duration expressions

PLEXIL expressions can have type `Date` or `Duration`. The former includes *time* and combined *date/time* expressions. Dates and durations are encoded as *strings* in the ISO-8601 format. Here are some examples of Date and Duration variable declarations.

```
Duration dur1;  // uninitialized duration variable
Date date1;     // uninitialized date variable
Duration dur2 = Duration("PT60M");  // 60 minutes
Date date2 = Date("2012-05-26T20:42:00.00Z");  // UTC time
Date date3 = "2011-12-03T00:42:12.00";  // local time
```

Dates and Durations are expressed as literals using the `Date` and `Duration` constructor, respectively. These are exemplified in the variable initializations shown above. Here are more examples:

```
// subtract 1.5 seconds from the given date.
date3 = date3 - Duration("PT1.5S");

// Calculate the duration between two dates.
dur2 = date3 - Date("2011-05-16T03:19:00");
```

Finally, here is a simple practical use of these types: an action that starts on or after a given date, and runs

for a specified duration:

```
Date Lookup time;
Date Lookup start;
Duration Lookup duration;

Test:
{
    Start Lookup(time, 1) >= Lookup(start);
    End   Lookup(time, 1) <= Self.EXECUTING.START + Lookup(duration);
}
```

Additional PLEXIL plans illustrating varied uses of dates and durations may be found in the directory `plexil/examples/temporal` in the PLEXIL source code distribution.

CAVEATS: At present, date and duration literals are not checked for valid syntax. Also, unspecified behavior will result if an arithmetic operation involving dates or durations yields a negative value.

Operations using Dates and Durations
The following arithmetic operations involving dates and durations are supported.

```
date       -   date     =  duration
date       +-  duration =  date
duration   +-  duration =  duration
duration   *   number   =  duration
duration   /   number   =  duration
duration   /   duration =  duration
duration   mod duration =  duration
duration   mod number   =  duration
abs duration            =  duration
```

Dates can be compared with the logical operators <, >, <=, >=, ==, and !=. So can Durations. Dates and Durations cannot be mixed with these operators.

Date and Duration representation
At present, dates and durations are not defined in Core PLEXIL. Recall that in Core PLEXIL, time is represented as a unitless real number, whose actual unit is application defined.

Expressions of type Date in the full PLEXIL language are translated into Core PLEXIL for execution, where they are converted to real numbers representing absolute time as *seconds* since the Unix epoch of Jan 1, 1970 (1970-01-01T00:00:00Z to be precise). This is a highly standard convention. At present, PLEXIL

does not support the use of alternate epochs.

Similarly, Duration expressions are converted into real numbers representing seconds.

CAVEAT: A key limitation in the current Plexil executive is that it does not recognize dates and durations as distinct from other real numbers. Therefore, for example, if date or duration values are inspected or printed in a running plan, a unitless real number will be shown. The PLEXIL team hopes to remedy this and make dates and durations better supported in general.

Arrays

PLEXIL provides just one aggregate data type, the *array*. At present, the array type in PLEXIL is somewhat limited compared to what's found in modern programming languages. PLEXIL arrays are homogenous and one-dimensional: a sequence of values of a single scalar data type, indexed by integers beginning with 0. Specifically, arrays may of type Integer, Real, String, or Boolean only.

PLEXIL provides both variables and literals of array type. Like other variables, array variables must be declared prior to use. An array declaration specifies its name, type, maximum size (number of elements), and, optionally, initial values for some or all of the array's elements. The memory needed by an array is allocated (for its maximum size) when the array is declared. Unlike scalar variables, array variables are *not* initialized to the Unknown value by default; rather, each element of the array is initialized to Unknown. Array indices start with 0.

The following examples illustrate the key properties of PLEXIL arrays.

```
Boolean flags[10];
```

This an array of ten booleans. Each element has the value Unknown (i.e. each element will fail the **isKnown** test).

```
Integer X[6] = #(1 3 5);
```

This example illustrates initialization of elements and the array literal. This array of 6 integers is initialized with an array containing 1,3, and 5 as its first three elements. That is, X[0] = 1, X[1] = 3, and X[2] = 5. The last three elements of X are Unknown. It is an error to initialize an array variable with an array containing more elements than its maximum size. (As an aside, the syntax for the array literal is taken from Common Lisp).

Arrays support the following operations. Assume an array named X.

- Read an element: `X[<index>]`, where <index> can be any integral expression. Array elements are a kind of expression, and thus may be used in any place where expressions are allowed.

- Assign an element: `X[<index>] = <expression>`. Assignments can occur only in assignment actions.
- Assign an entire array: `X = Y`, where Y is either an array variable or an array literal. It is an error if `Y` represents an array larger than `X`. If `Y` is smaller than `X` then the remaining elements in `X` will be filled with `Unknown`.
- Get the size of an array as an Integer: `arraySize(Y)`, where Y is an array-valued expression.
- Get the maximum size of an array as an Integer: `arrayMaxSize(Y)`, where Y is an array-valued expression.

Action State

A PLEXIL action can access its own internal state, or the internal state of other actions, but only those actions which are its siblings, children, or ancestors. (The internal state of more distant actions is not accessible).

Action state consists of:

- The current execution state of an action
- The start and end times of each state an action has encountered
- The outcome value of an action, if it has terminated
- The failure type of an action, if it has failed
- For command nodes, the last *command handle* received.

Each of these values is a unique type, with the exception of start and end times, which are of type `Date`. The only operations that can be performed with these values are comparison for equality or inequality with each other, or against a literal value.

The syntax for referencing these types of information is the following, where `<Id>` is the action's identifier.

```
<Id>.state
```

returns one of INACTIVE, WAITING, EXECUTING, FINISHED, ITERATION_ENDED, FAILING, FINISHING.

```
<Id>.<state>.<timepoint>
```

where `<state>` is one of the seven states listed above, and `<timepoint>` is one of START, END, will return the time elapsed (as a real number) since the given state started or ended (respectively) for the given action. If the requested timepoint has not occurred, the value of this variable is Unknown. For an explanation of time in PLEXIL, see the Overview.

```
<Id>.outcome
```

returns one of SUCCESS, FAILURE, or SKIPPED, if the given node has terminated (else it will return Unknown).

```
<Id>.failure
```

returns one of INVARIANT_CONDITION_FAILED, POST_CONDITION_FAILED, PRE_CONDITION_FAILED, PARENT_FAILED, if the node has terminated with failure (else it will return Unknown).

```
<Id>.command_handle
```

returns one of COMMAND_ACCEPTED, COMMAND_SUCCESS, COMMAND_RCVD_BY_SYSTEM, COMMAND_SENT_TO_SYSTEM, COMMAND_FAILED, or COMMAND_DENIED, if the node is executing (else it will return Unknown).

External State (Lookups)
External state is read through *lookups*. Lookups access states using domain-specific measurement names. The syntax for a lookup is:

```
Lookup(<state_name> [(<param>*)] [, <tolerance>])
```

where `<state_name>` is either an identifier or a string expression that evaluates to the desired state name. States can have parameters, which are specified by a comma-separated list of literal state names or string expressions that follow the state name. Tolerance, which is optional, must be a real number or real-valued variable; it specifies the granularity of accuracy for the lookup, and defaults to 0.0.

NOTE: For the state name, literal names are unquoted, while string expressions are parenthesized. For state parameters, literal names are double-quoted, while string expressions are given no special treatment.

Here are some basic examples:

```
Lookup(time)                       // queries the state named "time"

Lookup((pressureSensorName), 1.0)  // queries the state named by the
                                   // pressureSensorName variable

Lookup(At("rock1"))                  // queries the parameterized state At("rock1")
```

String expressions used for state names can include Lookup themselves. For example, here an external

query is used to get the name of a sensor for a Lookup:

```
Lookup((Lookup(ModuleVoltageSensorName("Crew Habitat"))), 0.1)
```

The tolerance parameter is optional and defaults to 0.0. If given, it must be a real number and specifies the minimum value by which the state must have changed since its last reading in order to be read again. The example above says to read the module voltage sensor when it changes at least 0.1. Tolerances are unitless in Plexil; the unit of measure they represent is specified by the queried external system. The tolerance parameter is meaningless and ignored in certain contexts. See the following section for an explanation.

Execution of Lookups

There are two contexts for lookups that are important to distinguish. One is the asynchronous context implied by an action's gate conditions (Start, Skip, End, Repeat). These conditons passively "wait" to become true. Lookups found in these conditions are processed as *subscriptions* to the external system for updates to the requested states. It is only in this context that *tolerance* is meaningful. These Lookup forms are compiled into *LookupOnChange* in Core PLEXIL's XML representation.

The second context for lookups is the synchronous context implied by an action's check conditions (Pre, Post, Invariant) and its body. In these contexts, a lookup is processed on demand, that is, its value is *fetched* at specific points in execution of the action. Tolerance is meaningless in this context, and ignored if specified. These Lookup forms are compiled into *LookupNow* in Core PLEXIL's XML representation.

The "time" state

The state name `time` is predefined in the Plexil executive. It returns the system time as a real number, which is compatible with the Date type. The units and epoch of the returned value are system dependent. On the typical platforms that support PLEXIL, they would be in *POSIX/Unix time*, i.e. the number of seconds since January 1, 1970 midnight UTC (1970-01-01T00:00:00.000Z).

Even though `time` is predefined, it must still be declared in the plan, in one of the following ways.

```
Date Lookup time;
Real Lookup time;
```

NOTE: Due to how the PLEXIL executive's interface to the system clock is implemented, a tolerance parameter is required for time lookups. E.g. to specify a tolerance of one time unit:

```
Lookup(time, 1)
```

Global Declarations

If a plan contains calls to system commands (i.e. command nodes), uses library actions, or queries world

state using lookups, these *must* be pre-declared.

These *global declarations* must occur first in Plexil files, before the top-level action. They can occur in any order; declarations for commands, lookups, and library actions can be intermixed freely.

Including global declarations as a standard practice has several advantages. First, it allows you to define and view your plan's entire external interface in one place, rather than having it scattered throughout the plan. Second, it enables static checking of your plan. Static checks will insure that your declarations are consistent and that all uses of declared items in the plan are correct.

The following are examples of global declarations.

```
// simple command
Command Stop();

// command with parameter (name is optional)
Command Drive(Real meters);

// command with return value
Boolean Command TakePicture(Integer, Integer, Real);

// state
Real Lookup Temperature;

// state with parameter
Boolean Lookup At (String location);

// library action
LibraryAction LibTest(In Real i, In vals[10], InOut Integer j);
```

Programming in PlexilPlexil Files

A file containing Plexil code can have any name, though its extension must be `.ple`. A Plexil file must contain exactly one construct, i.e. a single action. Your application may comprise many Plexil files; in this case, one file will contain the *top level* action, and the rest will contain library actions.

`plexilc`, the Plexil translator

Plexil files must be translated into XML for execution by the PLEXIL executive. Given a Plexil file `foo.ple`, translate it with the following command:

```
plexilc foo.ple
```

If `foo.ple` is free of errors, this command will create the Core PLEXIL XML file `foo.plx`.

An intermediate representation, `foo.epx` is likely to also be created. This file is ignored and may be deleted. (The `.epx` extension stands for *Extended Plexil*, a now obsolete name).

The `plexilc` program is also used to translate Test Executive simulation scripts and Plexilisp files. It supports the following command-line options (this list is obtainable by calling `plexilc` with no arguments):

```
-c, -check              Run static checker on output (only valid for plan files)
-d, -debug <logfile>    Print debug information to <logfile>
-h, -help               Print this help and exit
-o, -output <outfile>   Write translator output to <outfile>
-q, -quiet              Parse files quietly
-v, -version            Print translator version and exit
```

Some options are not supported by all source file formats.

If there are errors in your Plexil code, `plexilc` will report them, along with their line numbers and character positions. No output file is created. Often, when there are many errors, correcting one of them will take care of subsequent errors.

If `plexilc` outputs only *warnings* about your Plexil code, the translated output file is still created. Warnings usually indicate potentially serious errors in the program's logic, so they should be inspected and dealt with.