8

C-Scripts

C-Scripts provide a powerful and comfortable mechanism for implementing custom control blocks in the C programming language. They enable you to interact with the solver engine on a level very similar to that of built-in blocks.

Typical applications where C-Scripts are useful include:

- Implementing complex non-linear and/or piecewise functions. These would otherwise need to be modeled with complex block diagrams that are hard to read and maintain.
- Implementing modulators or pulse generators that require exact but flexible time step control.
- Incorporating external C code, e.g. for a DSP controller, into a simulation model.

There is no need to manually compile any code or even to install a compiler. A built-in compiler translates your C code on-the-fly to native machine code and links it dynamically into PLECS.

A detailed description of how C-Scripts work is given in the following section. For a quick start you can also have a look at the C-Script examples further below.

How C-Scripts Work

Since C-Scripts interact so closely with the solver engine, a good understanding of how a dynamic system solver works is advantageous. This is described in detail in the chapter “How PLECS Works” (on page 27).

C-Script Functions

A C-Script block, like any other control block, can be described as a mathematical (sub-)system having a set of inputs u, outputs y and state variables x_c, x_d that are related to each other by a set of equations:

$$\begin{aligned} y &= f_{\text{output}}(t, u, x_c, x_d) \\ x_d^{\text{next}} &= f_{\text{update}}(t, u, x_c, x_d) \\ \dot{x}_c &= f_{\text{derivative}}(t, u, x_c, x_d) \end{aligned}$$

A C-Script block has an individual code section for each of these functions and two additional sections for code to be executed at the start and termination of a simulation. The C code that you enter in these sections is automatically wrapped into C functions; the actual function interface is hidden to allow for future extensions. You can access block variables such as inputs, outputs and states by means of special macros that are described further below. The solver calls these C functions as required during the different stages of a simulation (see “Model Execution” on page 33).

Start Function

The start function is called at the beginning of a simulation. If the C-Script has continuous or discrete state variables, they should be initialized here using the macros `ContState(i)` and `DiscState(i)`.

Output Function

The output function is called during major and minor time steps in order to update the output signals of the block. The block inputs and outputs and the current time can be accessed with the macros `InputSignal(i, j)`, `OutputSignal(i, j)` and `CurrentTime`.

If you need to access any input signal during the output function call, you must check the **Input has direct feedthrough** box on the **Setup** pane of the C-Script dialog. This flag influences the block execution order and the occurrence of algebraic loops (see “Block Sorting” on page 31).

In general, output signals should be *continuous and smooth* during minor time steps; *discontinuities or sharp bends* should only occur during major time steps. Whether or not the call is made for a major time step can be inquired with the `IsMajorStep` macro. For details see "Modeling Discontinuities" below.

Note It is not safe to make any assumptions about the progression of time between calls to the output function. The output function may be called multiple times during the same major time step, and the time may jump back and forth between function calls during minor time steps. Code that should execute exactly *once* per major time step should be placed in the *update function*.

Update Function

If the block has discrete state variables, the update function is called once during a major time step after the output functions of all blocks have been processed. During this call, the discrete state variables should be updated using the `DiscState` macro.

Derivative Function

If the block has continuous state variables, the derivative function is called during the *integration loop* of the solver. During this call, the continuous state derivatives should be updated using the `ContDeriv` macro.

Derivatives should be *continuous and smooth* during minor time steps; *discontinuities or sharp bends* should only occur during major time steps. For details see "Modeling Discontinuities" below.

Terminate Function

The terminate function is called at the end of a simulation – regardless of whether the simulation stop time has been reached, the simulation has been stopped interactively, or an error has occurred. Use this function to free any resources that you may have allocated during the start function (e.g. file handles, memory etc.).

Store Custom State Function

This function is called at the end of a simulation before the terminate function. Use this function to store any custom values which are needed to restore the state of the C-Script. Continuous and discrete states are automatically stored. Use the macros `WriteCustomStateDouble`, `WriteCustomStateInt` and `WriteCustomStateData` to serialize your data.

Restore Custom State Function

This function is called after the start function and before the first call of the output function. The continuous and discrete states have already been restored at this point. Use this function to initialize your block with a previously stored custom state. Use the macros `ReadCustomStateDouble`, `ReadCustomStateInt` and `ReadCustomStateData` to deserialize your data and `SetErrorMessage` to report any issues to the user during restoring.

Code Declarations

This code section is used for global declarations and definitions (that is, global in the scope of the C-Script block). This is the place to include standard library headers (e.g. `math.h` or `stdio.h`) and to define macros, static variables and helper functions that you want to use in the C-Script functions.

You can also include external source files. The directory containing the model file is automatically added to the included search path, so you can specify the source file path relative to the model file.

Modeling Discontinuities

If the behavior of your C-Script block changes abruptly at certain instants, you must observe the following two rules in order to obtain accurate results:

1. If the time at which a discontinuity or event occurs is not known a priori but depends on the block inputs and/or states, you must define one or more zero-crossing signals, which aid the solver in locating the event. Failure to do so may result in a jitter on the event times.
2. During minor time steps, continuous state derivatives and output signals must be *continuous and smooth* functions. Failure to observe this may lead to gross numerical integration errors.

Defining Zero-crossing Functions

To define zero-crossing signals, register the required number of signals on the **Setup** pane of the C-Script dialog. In the output function, use the macro `ZCSignal(i)` to assign values to the individual zero-crossing signals depending e.g. on the block inputs or states or the current simulation time. The solver constantly monitors all zero-crossing signals of all blocks. If any one signal changes its sign during the current integration step, the step size is reduced so that the next major time step occurs *just after* the first zero-crossing. (See also "Event Detection Loop" on page 34.)

For instance, to model a comparator that must change its output when the input crosses a threshold of 1, you should define the following zero-crossing signal:

```
ZCSignal(0) = InputSignal(0, 0) - 1.;
```

Without the aid of the zero-crossing signal, the solver might make one step at a time when the input signal is e.g. 0.9 and the next step when the input signal has already increased to e.g. 1.23, so that the C-Script block would change its output too late.

With the zero-crossing signal, and provided that the input signal is continuous, the solver will be able to adjust the step size so that the C-Script output will change at the correct time.

Note If a zero-crossing signal depends solely on the simulation time, i.e. if an event time *is* known a priori, it is recommended to use a *discrete-variable sample time* and the `NextSampleHit` macro instead. (See "Discrete-Variable Sample Time" below.)

Keeping Functions Continuous During Minor Time Steps

The solver integrates the continuous state derivatives over a given interval (i.e. the current time step) by evaluating the derivatives at different times in the interval. It then fits a polynomial of a certain order to approximate the integral. (See also "Integration Loop" on page 34.) The standard Dormand-Prince solver, for instance, uses 6 derivative evaluations and approximates the integral with a polynomial of 5th order.

Obviously, the derivative of this polynomial is again a polynomial of one order less. On the other hand, to approximate a discontinuous or even just a non-smooth derivative function, a polynomial of infinite order would be required. This discrepancy may lead to huge truncation errors. It is therefore vital to describe the continuous state derivatives as *piecewise smooth* functions and make sure that only one subdomain of these functions is active throughout one integration step.

The output signal of a C-Script block might be used as the input signal of an integrator and thus might become the derivative of a continuous state variable. Therefore, output signals should be described as piecewise smooth functions as well.

Returning to the example of the comparator above, the complete output function code should look like this:

```
if (IsMajorStep)
{
  if (InputSignal(0, 0) >= 1.)
    OutputSignal(0, 0) = 1.;
  else
    OutputSignal(0, 0) = 0.;
}

ZCSignal(0) = InputSignal(0, 0) - 1.;
```

The condition `if (IsMajorStep)` ensures that the output signal can only change in major steps. It remains constant during the integration loop regardless of the values that the input signal assumes during these minor time steps. The zero-crossing signal, however, is also updated in minor time steps during the event detection loop of the solver.

Sample Time

A C-Script block can model a continuous system, a discrete system, or even a hybrid system having both continuous and discrete properties. Depending on which kind of system you want to model, you need to specify an appropriate **Sample time** on the **Setup** pane of the C-Script dialog. The sample time determines at which time steps (and at which stages) the solver calls the different C-Script functions.

Continuous Sample Time

Blocks with a continuous sample time (setting `0` or `[0, 0]`) are executed at every major and minor time step. You must choose a continuous sample time if

- the C-Script models a continuous (or piecewise continuous) function,
- the C-Script has continuous states or,
- the C-Script registers one or more zero-crossing signals for event detection.

Semi-Continuous Sample Time

Blocks with a semi-continuous sample time (setting `[0, -1]`) are executed at every major time step but not at minor time steps. You can choose a semi-continuous instead of a continuous sample time if the C-Script produces only discrete output values and does *not* need zero-crossing signals.

Discrete-Periodic Sample Time

Blocks with a discrete-periodic sample time (setting T_p or `[`T_p`,` T_o`]`) are executed at regularly spaced major time steps. The sample period T_p must be a positive real number. The sample offset T_o must be a positive real number in the interval $0 \leq T_\mathrm{o} < T_\mathrm{p}$; it may be omitted if it is zero.

The time steps, at which the output and update functions are executed, are calculated as $n \cdot T_\mathrm{p} + T_\mathrm{o}$ with an integer n.

Discrete-Variable Sample Time

Blocks with a discrete-variable sample time (setting `-2` or `[-2, 0]`) are executed at major time steps that are specified by the blocks themselves.

In a C-Script you assign the time, when the block should be executed next, to the macro `NextSampleHit`. This can be done either in the output or update function. At the latest, after the update function call, the `NextSampleHit` must be greater than the current simulation time. Otherwise, the simulation will be aborted with an error.

If a C-Script *only* has a discrete-variable sample time, the time of the first sample hit must be assigned in the start function. Otherwise, the C-Script will never be executed. During the start function, the simulation start time is available via the macro `CurrentTime`.

Note For discrete-variable sample times, PLECS Blockset can control the time steps taken by the Simulink solvers only indirectly by using an internal zero-crossing signal. Therfore, the actual simulation time at a discrete-variable sample hit may be slightly larger than the value that was specified as the next sample hit.

The solvers of PLECS Standalone, however, can evaluate the sample hit requests directly and are therefore guaranteed to meet the requests exactly.

Multiple Sample Times

If you want to model a hybrid system, you can specify multiple sample times in different rows of an $n \times 2$ matrix. For example, if your C-Script has continuous states but you must also ensure that it is executed every 0.5 seconds with an offset of 0.1 seconds, you would enter `[0, 0; 0.5, 0.1]`.

You can use the macro `IsSampleHit(i)` in the output and update functions in order to inquire which of the registered sample times has a hit in the current time step. The index `i` is a zero-based row number in the sample time matrix. In the above example, if your C-Script should perform certain actions only at the regular sampling intervals, you would write

```
if (IsSampleHit(1))
{
  // this code is only executed at t == n*0.5 + 0.1
}
```

To access the sample times during execution of the C-Script, use the macros `SampleTimePeriod(i)` and `SampleTimeOffset(i)`. In the case of inherited sample times, the actual resolved values are returned, not `[-1, 0]` (see “Sample Times” on page 38).

User Parameters

If you want to implement generic C-Scripts that can be used in different contexts, you can pass external parameters into the C functions.

External parameters are entered as a comma-separated list in the **Parameters** field on the **Setup** pane of the C-Script dialog. The individual parameters can be specified as MATLAB expressions and can reference workspace variables. They must evaluate to real scalars, vectors, matrices, 3d-arrays or strings.

Within the C functions you can inquire the number of external parameters with the macro `NumParameters`. The macros `ParamNumDims(i)` and `ParamDim(i, j)` return the number of dimensions of the individual parameters and their sizes. In the case of strings, `1` and the length of the string measured in C characters (`char`) is returned, respectively. Note that because the strings are UTF-8 encoded, the length returned by `ParamDim(i, j)` may be larger than the number of unicode characters in the string.

To access the actual parameter values, use the macro `ParamRealData(i, j)`, where `j` is a *linear index* into the data array. For example, to access the value in a certain row, column and page of a 3d-array, you write:

```
int rowIdx = 2;
int colIdx = 0;
int pageIdx = 1;
int numRows = ParamDim(0, 0);
int numCols = ParamDim(0, 1);
int elIdx = rowIdx + numRows*(colIdx + numCols*pageIdx);
double value = ParamRealData(0, elIdx);
```

To access string parameters, use the macro `ParamStringData(i)`. For example, to use the second parameter as an error message, you may write:

```
SetErrorMessage(ParamStringData(1));
```

Runtime Checks

If the box **Enable runtime checks** on the **Setup** pane of the C-Script dialog is checked, C-Script macros that access block data (e.g. signals values, states, parameters etc.) are wrapped with protective code to check whether an array index is out of range. Also, the C-Script function calls are wrapped with code to check for solver policy violations such as modifying states during minor time steps or accessing input signals in the output function without enabling direct feedthrough.

These runtime checks have a certain overhead, so once you are sure that your C-Script is free of errors you can disable them in order to increase the simulation speed. This is not recommended, however, because in this case access violations in your C-Script may cause PLECS to crash.

Note The runtime checks cannot guard you against access violations caused by direct memory access.

C-Script Examples

This section presents a collection of simple examples that demonstrate the different features of the C-Script and that you can use as starting points for your own projects. Note that the functionality of the example blocks is already available from blocks in the PLECS library.

A Simple Function – Times Two

The first example implements a block that simply multiplies a signal with 2. This block is described by the following system equation:

$$y = f_{\text{output}}(t, u, x_c, x_d) = 2 \cdot u$$

Block Setup The block has one input, one output, no states and no zero-crossing signals. It has direct feedthrough because the output function depends on the current input value. Since the output signal is continuous (provided that the input signal is) the sample time is also continuous, i.e. `[0, 0]` or simply `0`.

Output Function Code

```
OutputSignal(0, 0) = 2.*InputSignal(0, 0);
```

In every major and minor time step, the output function retrieves the current input value, multiplies it with 2 and assigns the result to the output.

Discrete States – Sampled Delay

This example implements a block that samples the input signals regularly with a period of one second and outputs the samples with a delay of one period. Such a block is described by the following set of system equations:

$$\begin{aligned} y &= f_{\text{output}}(t, u, x_c, x_d) &&= x_d \\ x_d^{\text{next}} &= f_{\text{update}}(t, u, x_c, x_d) &&= u \end{aligned}$$

Remember that in a major time step the solver first calls the block output function and then the block update function.

Block Setup The block has one input and one output. One discrete state variable is used to store the samples. The block does not have direct feedthrough because the input signal is not used in the output function but only in the update function. The sample time is `[1, 0]` or simply `1`.

Output Function Code

```
OutputSignal(0, 0) = DiscState(0);
```

Update Function Code

```
DiscState(0) = InputSignal(0, 0);
```

Continuous States – Integrator

This example implements a block that continuously integrates the input signal and outputs the value of the integral. Such a block is described by the following set of system equations:

$$\begin{aligned} y &= f_{\text{output}}(t, u, x_c, x_d) &&= x_c \\ \dot{x}_c &= f_{\text{derivative}}(t, u, x_c, x_d) &&= u \end{aligned}$$

Block Setup The block has one input and one output. One continuous state variable is used to integrate the input signal. The block does not have direct feedthrough because the input signal is not used in the output function but only in the derivative function. The sample time is continuous, i.e. `[0, 0]` or simply `0`.

Output Function Code

```
OutputSignal(0, 0) = ContState(0);
```

Derivative Function Code

```
ContDeriv(0) = InputSignal(0, 0);
```

Event Handling – Wrapping Integrator

This examples extends the previous one by implementing an integrator that wraps around when it reaches an upper or lower boundary (e.g. 2π and 0). Such an integrator is useful for building e.g. a PLL to avoid round-off errors that would occur if the phase angle increased indefinitely. This wrapping property can actually not be easily described with mathematical functions. However, the C code turns out to be fairly simple.

Block Setup The block has the same settings as in the previous example. Additionally, it requires two zero-crossing signals, in order to let the solver find the exact instants, at which the integrator state reaches the upper or lower boundary.

Output Function Code

```
#define PI 3.141592653589793
if (IsMajorStep)
{
  if (ContState(0) > 2*PI)
    ContState(0) -= 2*PI;
  else if (ContState(0) < 0)
    ContState(0) += 2*PI;
}
ZCSignal(0) = ContState(0);
ZCSignal(1) = ContState(0) - 2*PI;

OutputSignal(0, 0) = ContState(0);
```

In every major time step, if the integrator state has gone beyond the upper or lower boundary, 2π is added to or subtracted from the state so that it lies within the boundaries again. In every major and minor time step, the zero-crossing signals are calculated so that they become zero when the state is 0 resp. 2π. Finally, the integrator state is assigned to the output.

Note, that the state *must not* be modified during minor time steps, because then the solver is either itself updating the state (while integrating it) or trying to find the zeros of the zero-crossing functions, which in turn depend on the state. In either case an external modification of the state will lead to unpredictable results.

Derivative Function Code

```
ContDeriv(0) = InputSignal(0, 0);
```

Piecewise Smooth Functions – Saturation

This example implements a saturation block that is described by the following piecewise system equation:

$$y = f_{\text{output}}(t, u, x_c, x_d) = \begin{cases} 1, & \text{for } u \geq 1 \\ u, & \text{for } -1 < u < 1 \\ -1, & \text{for } 1 \leq u \end{cases}$$

When implementing this function, care must be taken to ensure that the active output equation does not change during an integration loop in order to avoid numerical errors (see "Modeling Discontinuities" on page 210).

Block Setup The block has one input, one output and no state variables. In order to make sure that a major step occurs whenever the input signal crosses the upper or lower limit, two zero-crossing signals are required.

Output Function Code

```
static enum { NO_LIMIT, LOWER_LIMIT, UPPER_LIMIT } mode;

if (IsMajorStep)
{
  if (InputSignal(0, 0) > 1.)
    mode = UPPER_LIMIT;
  else if (InputSignal(0, 0) < -1.)
    mode = LOWER_LIMIT;
  else
    mode = NO_LIMIT;
}

switch (mode)
{
  case NO_LIMIT:
    OutputSignal(0, 0) = InputSignal(0, 0);
    break;
  case UPPER_LIMIT:
    OutputSignal(0, 0) = 1.;
    break;
```

```
    case LOWER_LIMIT:
      OutputSignal(0, 0) = -1.;
      break;
  }

  ZCSignal(0) = InputSignal(0, 0) + 1.;
  ZCSignal(1) = InputSignal(0, 0) - 1.;
```

Ensuring that only one output equation will be used throughout an entire integration step requires a static mode variable that will retain its value between function calls. The active mode is determined in major time steps depending on the input signal. In the subsequent minor time steps, the equation indicated by the mode variable will be used *regardless of the input signal*.

If the step size were not properly limited and the input signal went beyond the limits during minor time steps, so would the output signal. This is prevented by the two zero-crossing signals that enable the solver to reduce the step size as soon as the input signal crosses either limit.

Note Instead of the static mode variable, a discrete state variable could also be used to control the active equation. In this particular application a static variable is sufficient because information needs to be passed only from one major time step to the subsequent *minor* time steps.

However, if information is to be passed from one major time step to a later *major* time step, a discrete state variable should be used, so that it can also be stored between multiple simulation runs.

Multiple Sample Times – Turn-on Delay

A turn-on delay is often needed for inverter controls in order to prevent short-circuits during commutation. When the input signal changes from 0 to 1, the output signal will follow after a prescribed delay time, provided that the input signal is still 1 at that time. When the input signal changes to 0, the output is reset immediately.

Block Setup The block has one input and one output. One discrete state variable is required to store the input signal value from the previous major time step.

Two sample times are needed: a semi-continuous sample time so that the input signal will be sampled at *every major time step*, and a discrete-variable sample time to enforce a major time step *exactly* after the prescribed delay time. The **Sample time** parameter is therefore set to `[0, -1; -2, 0]`.

As an additional feature the delay time is defined as an external user parameter.

Code Declarations

```
#include <float.h>
#define PREV_INPUT DiscState(0)
#define DELAY ParamRealData(0, 0)
```

The standard header file `float.h` defines two numerical constants, `DBL_MAX` and `DBL_EPSILON`, that will be needed in the output function. Additionally, two convenience macros are defined in order to make the following code more readable.

Start Function Code

```
if (NumParameters != 1)
{
  SetErrorMessage("One parameter required (delay time).");
  return;
}
if (ParamNumDims(0) != 2
    || ParamDim(0, 0) != 1 || ParamDim(0, 1) != 1
    || DELAY <= 0.)
{
  SetErrorMessage("Delay time must be a positive scalar.");
  return;
}
```

The start function checks whether the proper number of external parameters (i.e. one) has been provided, and whether this parameter has the proper dimensions and value.

Output Function Code

```
if (InputSignal(0, 0) == 0)
{
  OutputSignal(0, 0) = 0;
  NextSampleHit = DBL_MAX;
}
else if (PREV_INPUT == 0)
```

```
{
  NextSampleHit = CurrentTime + DELAY;
  if (NextSampleHit == CurrentTime)
    NextSampleHit = CurrentTime * (1.+DBL_EPSILON);
}
else if (IsSampleHit(1))
{
  OutputSignal(0, 0) = 1;
  NextSampleHit = DBL_MAX;
}
```

If the input signal is 0, the output signal is also set to 0 according to the block specifications. The next discrete-variable hit is set to some large number (in fact: the largest possible floating point number) because it is not needed in this case.

Otherwise, if the input signal is *not* 0 but it has been in the previous time step, i.e. if it just changed from 0 to 1, a discrete-variable sample hit is requested at `DELAY` seconds later than the current time.

Finally, if *both* the current and previous input signal values are nonzero and the discrete-variable sample time has been hit, i.e. if the delay time has just passed and the current input is still nonzero, the output is set to 1 and the next discrete-variable hit time is again reset to the largest possible floating point number.

The condition `if (NextSampleHit == CurrentTime)` requires special explanation: If `DELAY` is very small and the current time is very large, the sum of these two floating point numbers might again yield the current time value due to roundoff errors, which would lead to a simulation error. In this case the next sample hit is increased to the smallest possible floating point number that is still larger than the current time. Admittedly, this problem will only occur when the current time and the delay time are more than *15 decades* apart, and so it might be considered academic.

Update Function Code

```
PREV_INPUT = InputSignal(0, 0);
```

In the update function, the current input value is stored as the previous input value for the following time step.

Store Custom State Code

```
WriteCustomStateDouble(NextSampleHit);
```

The previous input value is stored automatically because it is a discrete state. The `NextSampleHit` has to be stored in the custom state.

Restore Custom State Code

```
NextSampleHit = ReadCustomStateDouble();
```

Restore the `NextSampleHit` value. When the simulation starts from a stored system state that was stored before this block was added to the schematic, the read operation will fail and PLECS reports a runtime error.

C-Script Macros

The following table summarizes the macros that can be used in the C-Script function code sections.

C-Script Data Access Macros

Macro	Type	Access	Description
`NumInputTerminals`	int	R	Returns the number of input terminals.
`NumOutputTerminals`	int	R	Returns the number of output terminals.
`NumInputSignals (int i)`	int	R	Returns the number of elements (i.e. the width) of the signal connected to the `i`th input terminal.
`NumOutputSignals (int i)`	int	R	Returns the number of elements (i.e. the width) of the signal connected to the `i`th output terminal.
`NumContStates`	int	R	Returns the number of continuous states.
`NumDiscStates`	int	R	Returns the number of discrete states.
`NumZCSignals`	int	R	Returns the number of zero-crossing signals.
`NumParameters`	int	R	Returns the number of user parameters.
`CurrentTime`	double	R	Returns the current simulation time (resp. the simulation start time during the start function call).

C-Script Data Access Macros (contd.)

Macro	Type	Access	Description
`NumSampleTime`	int	R	Returns the number of sample times.
`SampleTimePeriod (int i)`	int	R	Returns the period of the `i`th sample time.
`SampleTimeOffset (int i)`	int	R	Returns the offset of the `i`th sample time.
`IsMajorStep`	int	R	Returns `1` during major time steps, else `0`.
`IsSampleHit (int i)`	int	R	Returns `1` if the `i`th sample time currently has a hit, else `0`.
`NextSampleHit`	double	R/W	Specifies the next simulation time when the block should be executed. This is relevant only for blocks that have registered a discrete-variable sample time.
`InputSignal (int i, int j)`	double	R	Returns the value of the `j`th element of the `i`th input signal terminal. See C-Script block (see page 375) for information on how to increase the default number of input signal terminals.
`OutputSignal (int i, int j)`	double	R/W	Provides access to the value of the `j`th element of the `i`th output signal terminal. See C-Script block (see page 375) for information on how to increase the default number of output signal terminals. Output signals may *only* be changed during the output function call.
`ContState (int i)`	double	R/W	Provides access to the value of the `i`th continuous state. Continuous state variables may *not* be changed during minor time steps.
`ContDeriv (int i)`	double	R/W	Provides access to the derivative of the `i`th continuous state.
`DiscState (int i)`	double	R/W	Provides access to the value of the `i`th discrete state. Discrete state variables may *not* be changed during minor time steps.

C-Script Data Access Macros (contd.)

Macro	Type	Access	Description
`ZCSignal (int i)`	double	R/W	Provides access to the `i`th zero-crossing signal.
`ParamNumDims (int i)`	int	R	Returns the number of dimensions of the `i`th user parameter.
`ParamDim (int i, int j)`	int	R	Returns the `j`th dimension of the `i`th user parameter.
`ParamRealData (int i, int j)`	double	R	Returns the value of the `j`th element of the `i`th user parameter. The index `j` is a linear index into the parameter elements. Indices into multi-dimensional arrays must be calculated using the information provided by the `ParamNumDims` and `ParamDim` macros. If the parameter is a string, this macro will produce a runtime error or an access violation if runtime checks are disabled.
`ParamStringData (int i)`	char*	R	Returns a pointer to a UTF-8 encoded, null-terminated C string that represents the `i`th user parameter. If the parameter is not a string, this macro will produce a runtime error or returns `NULL` if runtime checks are disabled.

`WriteCustomStateDouble (double val)`
`WriteCustomStateInt (int val)`
`WriteCustomStateData (void *data, int len)`
	void
	void
	void
		W	Write a custom state of type double, int or custom data with `len` number of bytes. Use multiple calls for multiple values.

`ReadCustomStateDouble()`
`ReadCustomStateInt()`
`ReadCustomStateData (void *data, int len)`
	int
	double
	void
		R	Read a custom state of type double, int or custom data with `len` number of bytes. Use multiple calls for multiple values.

C-Script Data Access Macros (contd.)

Macro	Type	Access	Description
`SetErrorMessage (char *msg)`	void	W	Use this macro to report errors that occur in your code. The simulation will be terminated after the current simulation step. In general, this macro should be followed by a `return` statement. The pointer `msg` must point to static memory.
`SetWarningMessage (char *msg)`	void	W	Use this macro to report warnings. The warning status is reset as soon as the current C-Script function returns, so you do not need to reset it manually. The pointer `msg` must point to static memory.

Note The values of the input and output signals are not stored in contiguous memory. Therefore, signal values may only be accessed by using the macros, not by pointer arithmetic. For example, trying to access the second output using the following code will fail:

```
double *output = &OutputSignal(0, 0); // not recommended
output[1] = 1;     // fails
*(output + 1) = 1; // fails
OutputSignal(0, 1) = 1;     // ok
```

Note Prefer reading and writing of custom state variables with `double` and `int` values over `void*` custom data. The latter cannot handle the byte order (big endian, little endian) of your platform. To store a vector of doubles use the following code:

```
// platform independent code, recommended
WriteCustomStateInt(vectorSize);
for (int i = 0; i < vectorSize; ++i)
{
   WriteCustomStateDouble(vector[i]);
}

// platform dependent code, not recommended
WriteCustomStateData(&vectorSize, sizeof(vectorSize));
WriteCustomStateData(vector, vectorSize*sizeof(double));
```

Deprecated Macros

The macros `NumInputs`, `NumOutputs`, `Input(int i)` and `Output(int i)` are deprecated but are still supported for C-Scripts that have only a single input and output terminal.

9

State Machines

State machines are a formalism for event driven systems that move from one discrete state to another in response to discrete events. PLECS lets you graphically create and edit state machines using common concepts such as boxes for states and curved arrows for transitions, and simulate them together with a surrounding system. You can feed continuous or discrete signals into a state machine e.g. to react to external events and output discrete signals from a state machine e.g. as control signals. Actions are specified in the C programming language and can be associated with states and transitions. Thanks to their built-in timer events, state machines are equally useful for implementing supervisory controls and complex modulators.

This chapter is subdivided into three sections. The first section describes how you interact with the graphical editor to create and modify state machines. The second section describes the semantics of a state diagram and how they influence the execution of the state machine. The third section contains examples that highlight different features of the state machine.