8

C-脚本

C-脚本为在C编程语言中实现自定义控制模块提供了一种强大且便捷的机制。它们使您能够以与内置模块非常相似的方式与求解器引擎进行交互。

C-脚本在以下典型应用中非常有用：

- 实现复杂的非线性及/或分段函数。否则，这些函数需要通过复杂的模块图进行建模，而复杂的模块图难以阅读和维护。
- 实现调制器或脉冲发生器，这些设备需要精确但灵活的时间步长控制。
- 将外部C代码（例如，用于DSP控制器）集成到仿真模型中。

无需手动编译任何代码，甚至无需安装编译器。内置编译器会实时将您的C代码翻译成原生机器代码，并动态链接到PLECS中。

C-脚本的工作原理将在下一节中详细描述。如果您想快速上手，也可以查看下方提供的C-脚本示例。

C-脚本的工作原理

由于C-脚本与求解器引擎交互非常紧密，因此了解动态系统求解器的工作原理将大有裨益。这详细描述在“PLECS如何工作”章节中（第27页）。

C-脚本函数

C-Script模块，与其他控制模块一样，可以描述为一个具有一组输入 u、输出 y 和状态变量 x_c、x_d 的数学（子）系统，这些变量通过一组方程相互关联：

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

一个C-脚本模块为每个这些函数有一个独立的代码段，并且为在模拟的开始和结束时要执行的代码有两个额外的代码段。你输入到这些段中的C代码会被自动包装成C函数；实际的函数接口被隐藏以允许未来的扩展。你可以通过进一步描述的特殊宏访问模块变量，如输入、输出和状态。求解器在模拟的不同阶段按需调用这些C函数（参见第33页的“模型执行”）。

启动函数

启动函数在模拟开始时被调用。如果C-脚本有连续或离散状态变量，它们应该在这里使用宏 `ContState(i)` 和 `DiscState(i)`进行初始化。

输出函数

输出函数在主要和次要时间步期间被调用，以更新模块的输出信号。模块的输入和输出以及当前时间可以通过宏 `InputSignal(i, j)`, `OutputSignal(i, j)`和 `CurrentTime`进行访问。

如果您需要在输出函数调用期间访问任何输入信号，您必须在 C-脚本对话框的 **输入具有直通** 框中选中 **设置** 面板上的该选项。此标志会影响模块执行顺序和代数回路的产生（参见第 31 页的“模块排序”）。

通常，输出信号应在次要时间步期间保持 连续且平滑； 不连续性或锐利弯折 应仅在主要时间步期间发生。是否为主要时间步调用可以通过 `IsMajorStep` 宏查询。详情请参见下文的“建模不连续性”。

注意 在输出函数的调用之间对时间进程做出任何假设是不安全的。输出函数可能在同一个主要时间步内被调用多次，而在小时间步内，时间可能在函数调用之间来回跳跃。应该将应在每个主要时间步内精确执行一次 的代码放置在更新函数中。

更新函数

如果模块具有离散状态变量，更新函数在每个主要时间步内（在所有模块的输出函数处理完毕后）被调用一次。在此调用期间，应使用 `DiscState` 宏更新离散状态变量。

导数函数

如果模块具有连续状态变量，导数函数在求解器的积分循环 期间被调用。在此调用期间，应使用 `ContDeriv` 宏更新连续状态导数。

导数应在小时间步期间保持 连续和光滑； 不连续性或尖锐的弯折 应仅在大时间步期间出现。详情请参见下文“建模不连续性”。

终止函数

终止函数在模拟结束时被调用——无论模拟停止时间是否达到、模拟是否被交互式停止，或是否发生错误。使用此函数释放在启动函数期间（例如文件句柄、内存等）分配的任何资源。

存储自定义状态函数

此函数在模拟结束且终止函数被调用之前被调用。使用此函数存储任何需要恢复C-脚本状态的定制值。连续和离散状态会自动存储。使用宏 `WriteCustomStateDouble`、 `WriteCustomStateInt` 和`WriteCustomStateData` 来序列化您的数据。

恢复自定义状态函数

此函数在启动函数之后和输出函数第一次调用之前被调用。此时连续状态和离散状态已经重新存储。使用此函数用先前存储的自定义状态初始化你的模块。使用宏 `ReadCustomStateDouble`, `ReadCus-tomStateInt` 和 `ReadCustomStateData` 来反序列化你的数据，使用 `SetEr-rorMessage` 在恢复过程中向用户报告任何问题。

代码声明

此代码段用于全局声明和定义（即在C-脚本模块的作用域内全局）。这是包含标准库头文件（例如 `math.h` 或 `stdio.h`）并定义你希望在C-脚本函数中使用的宏、静态变量和辅助函数的地方。

您也可以包含外部源文件。包含模型文件的目录会自动添加到包含搜索路径中，因此您可以相对于模型文件指定源文件路径。

建模不连续性

如果您的C-Script模块在特定时刻的行为发生突然变化，为了获得准确的结果，您必须遵循以下两条规则：

1 如果不连续性或事件发生的时间事先未知，但取决于模块输入和/或状态，您必须定义一个或多个零交叉信号，这些信号有助于求解器定位事件。否则，事件时间可能会出现抖动。

2 在次要时间步中，连续状态导数和输出信号必须是 连续且平滑 的函数。若不遵守此规则，可能会导致严重的数值积分误差。

定义零交叉函数

要定义零交叉信号，请在C-脚本对话框的 **设置** 面板上注册所需数量的信号。在输出函数中，使用宏`ZCSignal(i)` 根据例如模块输入或状态或当前仿真时间，为各个零交叉信号分配值。求解器会持续监控所有模块的所有零交叉信号。如果在当前积分步中，任何信号改变其符号，则步长会减小，以便下一个主要时间步发生在 紧接 第一个零交叉之后。（另见第34页的“事件检测循环”）。

例如，要模拟一个当输入超过1的阈值时必须改变其输出的比较器，您应该定义以下零交叉信号:

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

没有 零交叉信号的辅助，求解器可能在输入信号为例如0.9时进行一步，然后在输入信号已经增加到例如1.23时进行下一步，因此C-脚本模块会过晚地改变其输出。

有了 零交叉信号，并且只要输入信号是连续的，求解器将能够调整步长，以便C-脚本输出能在正确的时间改变。

注意 如果一个零交叉信号完全依赖于仿真时间，即如果事件时间 是 预先已知的，建议使用一个`离散变量采样时间` 和 `NextSampleHit`宏。（见下文“离散变量采样时间”）。

在次要时间步期间保持函数连续性

该求解器通过在给定区间（即当前时间步）内的不同时间点评估导数来积分连续状态导数。然后，它将拟合一个特定阶数的多项式来近似积分。（参见第34页的“积分循环”。）例如，标准的Dormand- Prince求解器使用6次导数评估，并用5阶多项式来近似积分。

显然，该多项式的导数仍然是一个阶数较低的多项式。另一方面，要近似一个不连续的甚至只是非光滑的导数函数，需要无限阶的多项式。这种差异可能导致巨大的截断误差。因此，必须将连续状态导数描述为 分段光滑 函数，并确保在整个积分步中，这些函数只有一个子域是活跃的。

C-Script模块的输出信号可能被用作积分器的输入信号，从而可能成为连续状态变量的导数。因此，输出信号应描述为分段光滑函数。

回到上面比较器的示例，完整的输出函数代码应如下所示：

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

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

条件 `if (IsMajorStep)` 确保输出信号只能以主要步长变化。在积分循环期间，它保持不变，无论输入信号在这些次要时间步中取何值。然而，零交叉信号在求解器的事件检测循环中的次要时间步也会更新。

采样时间

一个C-Script模块可以模拟连续系统、离散系统，甚至具有连续和离散特性的混合系统。根据您想要模拟的系统类型，您需要在C-Script对话框的**采样时间**面板上指定一个适当的**设置**。采样时间决定了求解器在哪些时间步（以及哪些阶段）调用不同的C-Script函数。

连续采样时间

具有连续采样时间（设置0 或 `[0, 0]`）的模块在每次主要和次要时间步执行。如果您需要，必须选择连续采样时间。

- C-脚本模拟一个连续（或分段连续）函数，
- C-脚本具有连续状态或，
- C-脚本为事件检测注册一个或多个零交叉信号。

半连续采样时间

具有半连续采样时间（设置 `[0, -1]`）的模块在每个主要时间步执行，但在次要时间步不执行。如果 C-脚本仅产生离散输出值并且<样式 id='2'>不需要</样式>零交叉信号，您可以选择半连续采样时间而不是连续采样时间。

离散周期采样时间

具有离散周期采样时间的模块（设置 T_p 或 `[`T_p`,` T_o`]`）在等间隔的主要时间步执行。采样周期 T_p 必须是正实数。采样偏移 T_o 必须是区间 $0 \leq T_\mathrm{o} < T_\mathrm{p}$ 中的正实数；如果为零，则可以省略。

输出和更新函数执行的时间步计算为 $n \cdot T_\mathrm{p} + T_\mathrm{o}$ 与一个整数 n。

离散变量采样时间

具有离散变量采样时间的模块（设置 `-2` 或 `[-2, 0]`）在每个由模块自身指定的主要时间步执行。
在 C-脚本中，您将模块下次执行的时间分配给宏 `NextSampleHit`。这可以在输出或更新函数中完成。至少，在调用更新函数后， `NextSampleHit` 必须大于当前仿真时间。否则，模拟将因错误而中止。

如果一个C-脚本 仅 具有离散变量采样时间，则必须在启动函数中分配第一次采样命中的时间。否则，C-脚本将永远不会被执行。在启动函数期间，可以通过宏 `CurrentTime` 获取仿真开始时间。

注意 对于离散变量采样时间，PLECS模块集只能通过使用内部零交叉信号间接地控制Simulink求解器所采用的时间步。因此，在离散变量采样命中时的实际仿真时间可能比指定为下一次采样命中的值稍大。

然而，PLECS独立版的求解器可以直接评估采样命中请求，因此保证能够精确满足这些请求。

多个采样时间

如果您想对混合系统进行建模，可以在 $n \times 2$ 矩阵的不同行中指定多个采样时间。例如，如果您的C-脚本具有连续状态，但您还必须确保它每0.5秒执行一次，并具有0.1秒的偏移量，则您会输入 `[0, 0; 0.5, 0.1]`。

您可以在输出和更新函数中使用宏 `IsSampleHit(i)` 来查询在当前时间步中哪个已注册的采样时间有命中。索引 `i` 是采样时间矩阵中的零基行号。在上述示例中，如果您的C-脚本仅在常规采样间隔时执行某些操作，则您会编写

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

要访问 C-脚本 执行期间的采样时间，请使用宏 `SampleTimePeriod(i)` 和 `SampleTimeOffset(i)`。对于继承的采样时间，返回的是实际解析的值，而不是 `[-1, 0]` （参见第 38 页的“采样时间”）。

用户参数

如果您想实现可在不同上下文中使用的通用C脚本，可以将外部参数传递给C函数。

外部参数作为逗号分隔的列表输入到C-脚本对话框的**参数**字段中，该字段位于**设置**面板上。各个参数可以指定为MATLAB表达式，并且可以引用工作空间变量。它们必须评估为实标量、向量、矩阵、三维数组或字符串。

在C函数中，您可以使用宏 `NumParameters`查询外部参数的数量。宏 `ParamNumDims(i)` 和`ParamDim(i, j)` 分别返回各个参数的维度数量及其大小。对于字符串，`1` 和以C字符测量的字符串长度（`char`）分别被返回。请注意，由于字符串是UTF-8编码的， `ParamDim(i, j)` 返回的长度可能大于字符串中的unicode字符数。

要访问实际参数值，请使用宏 `ParamRealData(i, j)`，其中 `j` 是数据数组的线性索引 。例如，要访问三维数组中某一行、某一列和某一页的值，您编写：

```
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);
```

要访问字符串参数，请使用宏 `ParamStringData(i)`。例如，要将第二个参数用作错误消息，您可以这样写：

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

运行时检查

如果C-脚本对话框的<code>Setup</code>面板上的<code>Enableruntimechecks</code>复选框被选中，访问块数据（例如信号值、状态、参数等）的C-脚本宏将被保护代码包裹，以检查数组索引是否越界。此外，C-脚本函数调用将被包裹以检查求解器策略违规，例如在次要时间步修改状态或在输出函数中访问输入信号而不启用直接馈通。

这些运行时检查会带来一定的开销，因此一旦您确信您的C-脚本没有错误，您可以选择禁用它们以提高仿真速度。然而，不建议这样做，因为在这种情况下，您的C-脚本中的访问违规可能会导致PLECS崩溃。

注意 运行时检查不能防止直接内存访问引起的访问违规。

C-脚本示例

本节介绍一系列简单的示例，展示了C-脚本的不同功能，并可作为您自己项目的起点。请注意，示例模块的功能已通过PLECS库中的模块提供。

一个简单的函数——乘以二

第一个示例实现了一个简单的模块，该模块将信号乘以2。该模块由以下系统方程描述：

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

模块设置 该模块有一个输入、一个输出、无状态且无零交叉信号。它具有直接馈通，因为输出函数取决于当前输入值。由于输出信号是连续的（前提是输入信号也是），因此采样时间也是连续的，即 `[0, 0]`或简单地 `0`。

输出函数代码

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

在每个主要和次要时间步中，输出函数获取当前的输入值，将其乘以 2 并将结果赋值给输出。

离散状态 – 采样延迟

本示例实现了一个模块，该模块以一秒的周期定期采样输入信号，并以一个周期的延迟输出采样值。该模块由以下系统方程组描述：

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

记住，在一个主要时间步中，求解器首先调用模块输出函数，然后调用模块更新函数。

模块设置 该模块有一个输入和一个输出。使用一个离散状态变量来存储样本。该模块没有直接馈通，因为输入信号没有在输出函数中使用，而只在使用在更新函数中。采样时间是 `[1, 0]` 或简单地 `1`。

输出函数代码

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

更新函数代码

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

连续状态 – 积分器

这个例子实现了一个模块，它连续地积分输入信号并输出积分的值。这种模块由以下系统方程组描述：

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

模块设置 该模块有一个输入和一个输出。使用一个连续状态变量来积分输入信号。该模块没有直接馈通，因为输入信号没有在输出函数中使用，而只在使用在导数函数中。采样时间是连续的，即 `[0, 0]` 或简单地 `0`。

输出函数代码

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

导数函数代码

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

事件处理 – 包裹积分器

这个示例通过实现一个在达到上限或下限（例如 2π 和 0）时包裹的积分器来扩展上一个示例。这种积分器对于构建例如 PLL 以避免相位角无限增加时产生的舍入误差很有用。这种包裹特性实际上很难用数学函数来描述。然而，C 代码结果相当简单。

模块设置 该模块与上一个示例具有相同的设置。此外，它需要两个零交叉信号，以便求解器找到积分器状态达到上限或下限的确切时刻。

输出函数代码

```
#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);
```

在每一个主要时间步中，如果积分器状态超出了上限或下限， 2π 会被加到或从状态中减去，使其重新位于边界内。在每一个主要和次要时间步中，计算零穿越信号，以便当状态为0时，信号变为0 resp. 2π。最后，积分器状态被赋值给输出。

注意，在次要时间步期间，状态 不得 被修改，因为此时求解器要么正在更新状态（同时对其进行积分），要么试图寻找零交叉函数的零点，而这些零点又取决于状态。在任一情况下，对状态的外部修改都会导致不可预测的结果。

导函数代码

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

分段光滑函数 – 饱和

此示例实现了一个饱和模块，该模块由以下分段系统方程描述：

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

在实现此函数时，必须注意确保在积分循环期间活动输出方程不发生变化，以避免数值误差（参见第210页的“建模不连续性”）。

模块设置 该模块有一个输入、一个输出并且无状态变量。为了确保每当输入信号跨越上限或下限时都会发生一个主要步骤，需要两个零交叉信号。

输出函数代码

```
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.;
```

确保在整个积分步骤中仅使用一个输出方程，需要一个静态模式变量，该变量将在函数调用之间保持其值。活动模式是在主要时间步中根据输入信号确定的。在随后的次要时间步中，模式变量指示的方程将被使用 无论输入信号如何。

如果步长没有被正确限制，并且输入信号在次要时间步中超出限制，输出信号也会超出限制。这是通过两个零交叉信号来防止的，这些信号使求解器能够在输入信号跨越任一限制时立即减小步长。

注意 除了静态模式变量，还可以使用离散状态变量来控制活动方程。在这个特定的应用中，静态变量就足够了，因为信息只需要从一个主要时间步传递到随后的次要时间步。

然而，如果信息需要从一个主要时间步传递到后来的主要时间步，应该使用离散状态变量，以便它也可以在多次模拟运行之间进行存储。

多个采样时间 – 启动延迟

启动延迟通常需要在逆变器控制中加以应用，以防止换相期间发生短路。当输入信号从 0 变为 1 时，输出信号将在规定的延迟时间后跟随，前提是输入信号在该时间仍为 1。当输入信号变为 0 时，输出立即重置。

模块设置 该模块有一个输入和一个输出。需要一个离散状态变量来存储来自上一个主要时间步的输入信号值。

两个 采样时间需要：一个半连续采样时间，以便输入信号在每个主要时间步中被采样，以及一个离散变量采样时间以强制在规定的延迟时间后正好一个主要时间步。正好 因此，**采样时间** 参数被设置为 `[0, -1; -2, 0]`。

作为一个附加功能，延迟时间被定义为一个外部用户参数。

代码声明

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

标准头文件 `float.h` 定义了两个数值常量， `DBL_MAX` 和 `DBL_EPSILON`，这些常量将在输出函数中用到。此外，还定义了两个便利宏，以便使以下代码更易读。

启动函数代码

```
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;
}
```

启动函数检查是否提供了正确数量的外部参数（即一个），以及该参数是否具有正确的维度和值。

输出函数代码

```
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;
}
```

如果输入信号为0，则根据模块规范，输出信号也设置为0。下一个离散变量命中被设置为某个大数（实际上：最大的浮点数），因为它在这种情况下不需要。

否则，如果输入信号是 非 0 但它在前一时间步中已经存在，即如果它刚刚从 0 变为 1，则需要在当前时间之后 `DELAY` 秒请求离散变量采样命中。

最后，如果 两者 当前和前一个输入信号值都是非零，并且离散变量采样时间已被命中，即如果延迟时间刚刚过去且当前输入仍然非零，则输出设置为 1，下一个离散变量命中时间再次重置为最大可能的浮点数。

条件 `if (NextSampleHit == CurrentTime)` 需要特别解释：如果 `DELAY` 非常小且当前时间非常大，由于舍入误差，这两个浮点数的和可能再次得到当前时间值，这将导致模拟错误。在这种情况下，下一个采样命中增加到仍然大于当前时间的最小可能的浮点数。诚然，这个问题只会在当前时间和延迟时间相差超过 *15* 数量级时发生，因此可能被认为具有学术性。

更新函数代码

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

在更新函数中，当前输入值被存储为下一个时间步的先前输入值。

存储自定义状态代码

```
WriteCustomStateDouble(NextSampleHit);
```

由于它是离散状态，之前的输入值会自动存储。 `NextSampleHit` 必须存储在自定义状态中。

恢复自定义状态代码

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

恢复 `NextSampleHit` 值。当模拟从添加此模块到原理图之前存储的系统状态开始时，读取操作将失败，PLECS报告运行时错误。

C脚本宏

下表总结了可以在C脚本函数代码部分使用的宏。

C脚本数据访问宏

宏	Type	访问	描述
`NumInputTerminals`	int	R	返回输入终端的数量。
`NumOutputTerminals`	int	R	返回输出终端的数量。
`NumInputSignals (int i)`	int	R	返回连接到第 `i` 个输入终端的信号所包含的元素数量（即宽度）。
`NumOutputSignals (int i)`	int	R	返回连接到第 `i` 个输出终端的信号所包含的元素数量（即宽度）。
`NumContStates`	int	R	返回连续状态的数量。
`NumDiscStates`	int	R	返回离散状态的数量。
`NumZCSignals`	int	R	返回零交叉信号的数量。
`NumParameters`	int	R	返回用户参数的数量。
`CurrentTime`	double	R	返回当前仿真时间（或：在启动函数调用期间仿真开始时间）。

C-脚本数据访问宏 (续)

宏	Type	访问	描述
`NumSampleTime`	int	R	返回样本时间数。
`SampleTimePeriod (int i)`	int	R	返回第 `i` 个采样时间的周期。
`SampleTimeOffset (int i)`	int	R	返回第 `i` 个采样时间的偏移量。
`IsMajorStep`	int	R	在大时间步期间返回 `1`，否则返回 `0`。
`IsSampleHit (int i)`	int	R	如果当前的 ith 采样时间有命中，则返回 `1`，否则返回 `0`。
`NextSampleHit`	double	R/W	指定模块应在何时执行的下一次仿真时间。这仅与已注册离散变量采样时间的模块相关。
`InputSignal (int i, int j)`	double	R	返回 `j` 个输入信号终端的第 `i` 个元素的值。有关如何增加默认输入信号终端数量的信息，请参阅 C-Script 模块（参见第 375 页）。
`OutputSignal (int i, int j)`	double	R/W	提供对 `j` 个输出信号终端的第 `i` 个元素的值的访问。有关如何增加默认输出信号终端数量的信息，请参阅 C-Script 模块（参见第 375 页）。输出信号只能 在输出函数调用期间更改。
`ContState (int i)`	double	R/W	提供对第 `i` 个连续状态的值的访问。连续状态变量在次要时间步期间可能 不能 被更改。
`ContDeriv (int i)`	double	R/W	提供对第 `i` 个连续状态的导数的访问。
`DiscState (int i)`	double	R/W	提供对第 `i` 个离散状态值的访问。离散状态变量在次要时间步期间可能 不能 被更改。

C脚本数据访问宏 (续)

宏	Type	访问	描述
`ZCSignal (int i)`	double	R/W	提供对第 i个零交叉信号的访问。
`ParamNumDims (int i)`	int	R	返回第 i个用户参数的维度数量。
`ParamDim (int i, int j)`	int	R	返回第 j个第 i个用户参数的维度。
`ParamRealData (int i, int j)`	double	R	返回第 j个用户参数的第 i个元素的值。索引 j 是参数元素的线性索引。多维数组的索引必须使用 ParamNumDims 和ParamDim 宏提供的信息来计算。如果参数是字符串，当运行时检查被禁用时，此宏将产生运行时错误或访问违规。
`ParamStringData (int i)`	char*	R	返回一个指向UTF-8编码、空终止的C字符串的指针，该字符串表示第 i个用户参数。如果参数不是字符串，当运行时检查被禁用时，此宏将产生运行时错误或返回NULL 。

`WriteCustomStateDouble (double val)`
`WriteCustomStateInt (int val)`
`WriteCustomStateData (void *data, int len)`
	void
	void
	void
		W	编写类型为 double、int 或自定义数据的自定义状态，使用 len 字节数。对于多个值，请使用多次调用。

`ReadCustomStateDouble()`
`ReadCustomStateInt()`
`ReadCustomStateData (void *data, int len)`
	int
	double
	void
		R	读取类型为 double、int 或自定义数据的自定义状态，使用 len 字节数。对于多个值，请使用多次调用。

C-脚本数据访问宏 (续)

宏	Type	访问	描述
`SetErrorMessage (char *msg)`	void	W	使用此宏报告您代码中发生的错误。模拟将在当前模拟步骤后终止。通常，此宏应后跟一个 `return` 语句。指针 `msg` 必须指向静态内存。
`SetWarningMessage (char *msg)`	void	W	使用此宏报告警告。警告状态在当前 C-脚本函数返回后立即重置，因此您无需手动重置。指针 `msg` 必须指向静态内存。

注意 输入和输出信号的值不是连续存储在内存中的。因此，信号值只能通过使用宏来访问，而不能通过指针算术。例如，尝试使用以下代码访问第二个输出将失败：

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

Note 优先使用 `double` 读写自定义状态变量
`int` 值超过 `void*` 自定义数据。后者无法处理字节序（大
字节序（如小端字节序）的平台。要存储一个double向量，请使用...
以下代码：

```
// 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));
```

已弃用的宏

宏 `NumInputs`, `NumOutputs`, `Input(int i)` 和 `Output(int i)` 已弃用，但仍然支持仅具有单个输入和输出终端的C-脚本。

9

状态机

状态机是一种用于事件驱动系统的形式化方法，它通过离散事件从一个离散状态转移到另一个离散状态。PLECS允许您使用状态框和带曲线路径的转换等常见概念图形化创建和编辑状态机，并与外围系统一起模拟它们。您可以将连续或离散信号输入状态机，例如以响应外部事件，并将离散信号从状态机输出，例如作为控制信号。动作在C编程语言中指定，并且可以与状态和转换相关联。由于它们内置了计时器事件，状态机同样适用于实现监督控制和复杂调制器。

本章分为三个部分。第一部分描述了如何使用图形编辑器与状态机交互以创建和修改状态机。第二部分描述了状态图的语义以及它们如何影响状态机的执行。第三部分包含示例，突出显示了状态机的不同功能。