

The Mitrion Command Line Interface

```
mitrion [options] myprog.mtc
```

Options (for more options, see SDK manual) :

```
-h -help           Show help message
-version          Show version numbers
-cpp             First run cpp on source-file
-gui            Show simulator/debugger
-batch          Run simulator in batch mode
-input <file_1.. file_n> Input datafiles to simulator
-mem-io         Printout at every memory access
-output <basename> Basename of output from simulator
-configure      Configure a Mitrion Virtual Processor
-platform <name> Select platform for VHDL output
-sizes          Show FPGA usage
-vhdl-filename <file> Override default VHDL file name
-logfile <file> Write printout to file
-log-level <number> Set log level to: 0-8
-log <levestring!> Set log level to: NONE, ALWAYS,
INTERNAL, ERROR, WARNING,
PHASE, SUMMARY, RUNTIME or
INTERNALWARNING
--             End of options, arguments after '--'
              are source files)
```

Types

```
uint: bitwidth           unsigned integer
int: bitwidth           signed integer
bool                    boolean
float: mantissa.exp     floating point number
bits: bitwidth         raw binary number
type [ size ]           vector
type < size >           list
type (. .)               stream
tuple{type,type,...}    tuple
```

Lists, vector and stream constructors

```
Direct constructor list: < var, var, ..., var >
Direct constructor vector: [ var, var, ..., var ]
Direct constructor stream: (. var, var, ..., var .)
Direct constructor empty stream: (. - .)
Range constructor list: < const..const >
Range constructor vector: [ const..const ]
Range constructor stream: (. const .. const .)
```

Memory

```
mem it1 = memcreate(mem int:32[256] it_last);
(val, it2) = memread(it1, idx);
it3 = memwrite(it2, idx+1, val);
it_last = it3;
```

Language constructs

Mitrion-C 1.5;

```
result-collection = foreach(element-declare[, element-declare, ...] in collection-expression[,
collection-expression, ...] [by index-declare]) body expression|block;
```

```
variable | (variable[, variable, ...]) = for([element-declare[, element-declare, ...] in] collection-expression[,collection-expression, ...])
{
    body statements;
} [>] return-var | ([>] return-var[, [>] return-var, ...]);
```

```
variable | (variable[, variable, ...]) = while (conditional) [iterations number]
{
    body statements;
} [>] return-var | ( [>] return-var[, [>] return-var, ...]);
```

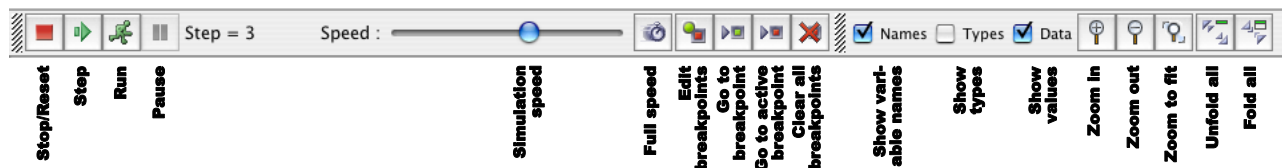
```
variable | (variable[, variable, ...]) = if (conditional)
    body expression|block
else
    body expression|block;
```

```
watch variable;
```

Function Definition

```
[return-types] function-name([type] var [, [type] var, ...]) body-expression|block;
```

Mitrion Simulator Controls



foreach

Multiplying each element in a list by its position in the list:

```
int:7<100> list = <1..100>;
int:16<100> a = foreach (int:7 elmnt in list by index)
{
    int:16 new_elmnt = elmnt * index;
} new_elmnt;
```

Multiplying two vectors:

```
int:7[100] a = [1..100];
int:7[100] b = [2..101];

int:14[100] c = foreach (elmnt_a, elmnt_b in a, b)
    elmnt_a * elmnt_b;
```

for

Calculating both the sum-total and running total of a list:

```
int:16 partSum = 1;
(int:16, int:16<10>) (sum, run) = for(i in <1..10>)
{
    partSum = partSum + i;
} (partSum, >> partSum);
```

while

Babylonian method single precision square root:

```
sqr_bab (float:24.8 n) {
    const float:24.8 ZERO = 0.0;
    float:24.8 x = 1.0;
    float:24.8 e = 1.0;
    res = while (abs(e) > ZERO) {
        new_x = 0.5 * (x + n / x);
        e = x - new_x;
        x = new_x;
    } x;
} res;
```

if

Counting the number of prime-numbers less than 10000:

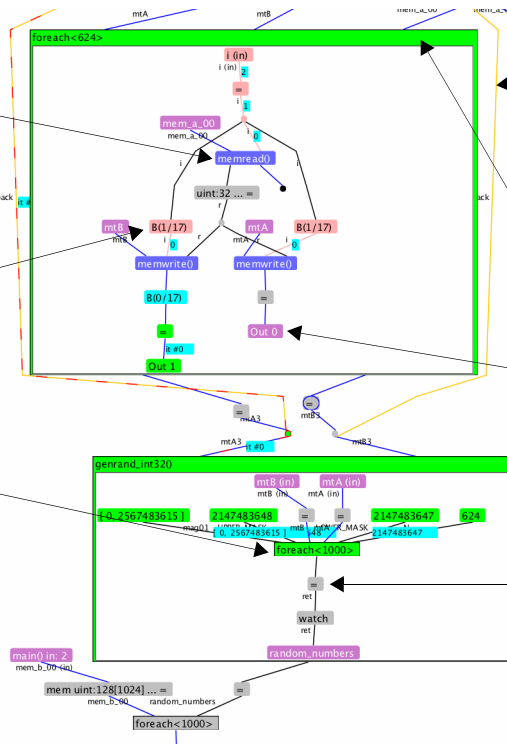
```
int:13 currentCount = 0;
int:13 n = for(index in <0..9999>)
{
    currentCount = if(isPrime(index))
        currentCount + 1
    else
        currentCount
};
```

Data Dependency Graph Color Legend

Blue edges and nodes: Blue nodes operate on instance tokens. Blue edges mark a relationship on an instance token. All memory access nodes for a specific RAM-bank are connected by blue edges.

Red nodes: Red nodes appear during simulation when a node cannot complete its operation, because the output was not able to be consumed. The edges connecting the node to blocking receiving nodes are also red, dashed red and blue, or dashed red and yellow.

Nodes with a black border: Nodes that in turn contain sub-graphs. Click on them to unfold them. (Right-click for recursive unfold.)



Yellow edges: Edges that have a reversed dependency, the node above the edge is dependent on the node below. These edges close dependency loops.

Green nodes: A node that has performed its operation in the last simulation step.

Purple nodes: Inputs and outputs to sub-graphs. If the compiler is unable to name the input or output, the node is labeled **In X** or **Out X** where X is the position of the input or output from the left edge of the surrounding node.

Grey nodes: Waiting node. Nodes that are currently not performing any operation due to lack of data.