

Embedded Scheme->C — 1 February 1993

Joel F. Bartlett

One of the major goals of Scheme->C was to build a Scheme compiler and runtime system that could coexist with other programming languages. While this effort has been quite successful, Scheme->C has not offered everything needed by applications that wish to embed a Scheme server, feed it arbitrary expressions for evaluation, yet remain responsive to additional requests.

For example, a database server might want to evaluate arbitrary Scheme expressions to verify record update operations, record access rights, or provide data that is computed rather than being resident in the database. Or, an event driven application for a PC or a Macintosh might like to embed Scheme.

In order to safely execute in such a variety of environments, the Scheme system must allow the application to handle external events on a timely basis and place minimum requirements on the available system services. In order to solve these problems, the 01feb93 release of Scheme->C has been modified in the following areas.

Explicit Time Slicing: In order to return to the caller at regular intervals, Scheme->C can be compiled with explicit time slicing. On each procedure entry or backwards branch, a counter is decremented. When the counter goes to 0, Scheme returns control to the application program at the point where it was called. At a later time, the application has the option of continuing the previous computation or starting a new computation.

Explicit Stack Overflow Checks: Scheme->C may be compiled with explicit stack overflow checks. This is necessary as many environments have no other means for detecting a stack overflow which could damage the embedding application or crash the personal computer.

Request-response interaction: When used as an embedded server, all interaction with Scheme is via one interface procedure. Errors and breakpoints are handled across this interface like any other type of request.

Operating System independence: Unlike previous releases, Scheme->C does not assume the existence of a UNIX-like I/O system. Rather than directly calling the host I/O system, all requests are via implementation specific routines in `scrt/cio.c`. In fact, an embedded Scheme->C server doesn't assume that the client

even has an I/O system. Instead the stdout and stderr ports are string output ports.

Traps: Previous releases of Scheme->C used operating system traps to detect division by zero and keyboard interrupts. Division by zero errors are now explicitly tested for, and keyboard interrupt signals are not used by embedded Scheme->C systems.

Application interface

Applications evaluate a Scheme expression by calling the procedure `scheme2c`:

```
void  scheme2c( char *expression,
                int  *status,
                char **result,
                char **error )
```

where `expression` is a pointer to a null terminated string of ASCII characters that is the Scheme expression to evaluate. When the procedure returns, the result is stored in `status`, `result`, and `error`. The value returned in `status` is interpreted as follows:

- 0: expression evaluated normally. The value is saved in `*SCHEME2C-RESULT*` within the Scheme system and it is also written to Scheme's stdout-port.
- 1: an error occurred. The error message is written to Scheme's stderr-port. If no previous error is being examined, the stack trace is written to Scheme's stderr-port and the associated environments are in the list `*ERROR-ENV*`. The client should evaluate `(RESET-ERROR)` when done examining the error state. Note that if additional errors occur before `(RESET-ERROR)` is evaluated, they will not cause a stack dump, nor have the error environment saved.
- 2: an internal error in Scheme->C occurred. The error message is reported via Scheme's stderr-port. No further execution is possible.
- 3: the computation timed out. Evaluate `(PROCEED)` to continue execution. Evaluate `(PROCEED?)` to cause a breakpoint when execution resumes.
- 4: a procedure entry breakpoint occurred. The call arguments are written to Scheme's stderr-port and the associated environments are in the list `*BPT-ENV*`. The procedure stack trace can be viewed by evaluating `(BACKTRACE)`. The procedure arguments are in `*ARGS*`. Evaluate `(PROCEED)` to continue execution, or `(RESET-BPT)` to abort.
- 5: a procedure exit breakpoint occurred. The result is written to Scheme's stderr-port and saved in `*RESULT*`. The environments are in the list `*BPT-ENV*`.

Evaluate `(PROCEED)` to continue execution, `(PROCEED expression)` to continue returning a new value, or `(RESET-BPT)` to abort. Note that additional breakpoints will not occur while examining the state of a breakpoint.

The value returned in `result` is pointer to a null terminated string of ASCII characters that is the contents of Scheme's `stdout-port`, i.e. the standard output port. The value returned in `error` is a pointer to a null terminated string of ASCII characters that is the contents of Scheme's `stderr-port`, i.e. the error output port.

The stack size which is used by an embedded Scheme system is set by evaluating `(set-stack-size! expression)`, where *expression* is the size in bytes. The current stack size can be obtained by evaluating `(stack-size)`. Scheme reserves a portion of this stack for error recovery, but this may be exceeded on some implementations. An application should verify that it has set the stack correctly by evaluating an expression that forces a stack overflow error and then verifying that the Scheme stack has not overflowed into the application.

The Scheme time slice is set by evaluating `(set-time-slice! expression)`, where *expression* is the number of Scheme procedure calls that should be made in a time slice. Experiment to find the right value for your application. The current time slice value is obtained by evaluating `(time-slice)`.

Sample embedded application

A sample embedded Scheme application, `embedded`, is found in the directory `scrt`. A typescript of its execution shows how an application should interact with an embedded Scheme system.

```
$ embedded
Embedded Scheme->C Test Bed
0- (+ 1 2)
3
0-
```

The program prompts the user for a Scheme expression which is then evaluated using `scheme2c`. The `result` and `error` messages are then printed, followed by a prompt (incorporating the value of `status`) for the next expression.

```
0- (time-slice)
100000
0- (stack-size)
57000
0- (let loop ((i 0)) (loop (+ i 1)))
3- (proceed)
```

```

3- (proceed)
3- (proceed?)
(+ I 1) in ENV-0
(LOOP (+ I 1)) in ENV-1
(EVAL ...)
(EXECUTE [inside SCHEME2C] ...)
(SCREP_SCHEME2C ...)
4-

```

After obtaining the current time slice and stack size values, a looping expression was entered. Twice it timed out and was continued by evaluating the expression (proceed). The third time it timed out, (proceed?) was entered to force a breakpoint.

```

4- (list-ref *bpt-env* 0)
((I . 21345) (LOOP . #*PROCEDURE*) ($_0 . 0))
0- (proceed)
3- (proceed?)
(LOOP (+ I 1)) in ENV-0
(EVAL ...)
(EXECUTE [inside SCHEME2C] ...)
(SCREP_SCHEME2C ...)
4- (list-ref *bpt-env* 0)
((I . 28476) (LOOP . #*PROCEDURE*) ($_0 . 0))
0- (reset-bpt)
#F
0-

```

The environment at the time of the breakpoint was examined to find the value of *i* by looking at element 0, corresponding to env-0, of the list **bpt-env**. The program was then continued by (proceed) and then *i* was examined at the end of the next time slice to find out how much work was done.

```

0- (let ((x 1) (y 2)) (+ x y z))
***** Z Top-level symbol is undefined
(+ X Y Z) in ENV-0
(EVAL ...)
(EXECUTE [inside SCHEME2C] ...)
(SCREP_SCHEME2C ...)
1- *bptenv*
***** *BPTENV* Top-level symbol is undefined
0- *error-env*
(((Y . 2) (X . 1)))
0- (reset-error)

```

```
#F
0-
```

An error occurred and then error environment was examined. While examining the error environment, another error occurred. This simply resulted in an error message.

```
0- (define (f x) (* 2 x))
F
0- (bpt f)
F
0- (f 23)
0 -calls - (F 23)
4- (proceed)
0 -returns- 46
5- (proceed)
46
0- (f 40)
0 -calls - (F 40)
4- (f 10)
20
0- (proceed)
0 -returns- 80
5- (proceed 15)
15
0-
```

The final example shows the use of procedure breakpoints. One thing to note, is that breakpoints do not nest.

Adding Your Code to an Embedded Scheme->C System

There are two ways to do this. The easiest is to replace the module `scrtuser` (in the files `scrtuser.sc` and `scrtuser.c`) and then rebuild the embedded Scheme system. An alternative is to separately compile your modules, link them into your application, and then explicitly initialize your modules after Scheme has been initialized. This is done by calling `scheme2c` with an expression like `"#t"` and then calling your module initialization procedures which have the form *module-name__init*.