C Runtime In Tcl

Andreas Kupries Vancouver, BC CA
akupries@shaw.ca

ABSTRACT

This paper shows off and demonstrates a number of major features and supporting packages which were added to `Critcl` since its inception.

1. INTRODUCTION

While `Tcl` [6] is not only an easy to use language, but often also fast enough, sometimes it is not enough.

Out of this desire for performance the "C Runtime In Tcl" was born, `Critcl` [22] in short. Initially conceived and maintained by Jean-Claude Wippler [12] the latter task came to me[11] after a time, via Steve Landers [15].

Note that Steve also wrote a very good introductory paper[10] to Critcl in 2002 and presented it at that year's Tcl conference. As about 80% of that paper is still true today this paper will not belabor the point and simply concentrate on the changes `Critcl` underwent since then.

The structure of the paper is this. First the next section will provide an overview of the various usage modes of Critcl and how they changed. This is followed by a section outlining the changes and extensions to the core API, and then after that a section explaining the various supporting packages which make a number of things more convenient or easy to do. This is followed by a section listing incompatibilites not fitting anywhere else, and thoughts about future development.

The code used in the demonstrations and examples is pulled from various packages using `Critcl`, specifically `CRIMP` [21], `TclYAML` [32], `TclLinenoise` [27], `KineTcl` [26], and `Marpa` [30].

2. USAGE MODES

`Critcl` started out with a single mode of operation, the "compile & run". In this mode `Critcl` is used as a package which collects the C fragments embedded in the Tcl code in memory and arranges with the `Tcl`'s auto-loader to compile and load them when needed. A cache directory is used to keep the resulting binaries between sessions, to reduce the amount of time spent on compilation further.

Due to the big disadvantage of the above, namely the need for a usable C compiler at runtime, very likely in a production environment, a pre-compilation mode was quickly added, to compile the C code for distribution once and then simply use the results at runtime, without the need for a compiler at that point. The entrypoint for these was the then-new `critcl` application.

Actually there were two such build modes, one resulting in just a plain shared libary (*-lib*), and the other in a proper, installable package (*-pkg*).

That was the state in 2002.

Since then the *-lib*-mode got removed, as nobody really used it.

At the same time (Oct 2011, v3.0 release) a new "conversion" mode was added instead, *-tea*. As the name of the option (hopefully) implies, this mode takes the Tcl code, embedded C code, etc. of the package and wraps them into the machinery expected of a regular C extension, i.e. a TEA-compatible combination of `configure` and `Makefile`.

This was and is intended for automatic package build systems with strict requirements on the API between them and the package to build.

In the case of ActiveState's[5] build system it was in the end easier to extend it to be able to detect `Critcl`-based packages and build them directly. In a similar vein my own `Kettle` [3] does know how to handle them, as does Sean Wood's[17] `PracTcl` [31]. `BAWT` [1] is able to use `Critcl` since version `0.3.0`, recently released. The state of `Quill` [4], and `kbskit` [2] with respect to `Critcl` is not known.

As such the *-tea* mode looks to be an experiment which failed. That said, the mode still exists, if somebody wishes to play with it.

3. API CHANGES

Beyond the new and changed modes a lot of new things were added, both in the core API, and via supporting packages. Among these are:

1. Better support for package metadata
2. Stubs table support
3. Optional and variadic arguments to `cproc`
4. Extended type support
5. Compiler diagnostic support
6. More efficient string usage via string pools
7. Improved enumeration support
8. Bitmap support
9. Classes and objects

The following subsections and the next section with its subsections will describe them all, in detail.

3.1 Meta Data

When still working at ActiveState one of the things we needed to support the `TEApot` repository was meta data for packages, i.e. package descriptions with keywords, categorization, etc.

While I was not that sucessful in promoting the use to package authors[1] as a maintainer of `Critcl` I was able to add meta-data support into it.

`::critcl::license` *author* ?*text*...?	Specify author and license
`::critcl::summary` *text*	Specify short description of the package
`::critcl::description` *text*	Specify a longer description
`::critcl::subject` ?*key*...?	Specify keywords and -phrases for an index
`::critcl::meta` *key* ?*word*...?	Specify arbitrary meta-data
`::critcl::meta?` *key*	Return stored meta-data for a key
`::critcl::buildrequirement` *script*	Hide package dependencies from the meta data

Table 1: Meta-Data Declaration Commands

Listing 1: `KineTcl` meta data declarations

```
critcl::license \
    {Andreas Kupries} \
    {Under a BSD license.}

critcl::summary \
    {OpenNI based Tcl binding to Kinect and similar sensor systems}

critcl::description {
    This package provides access to Kinect and similar sensor system,
    through binding to the OpenNI framework.
}

critcl::subject kinect primesense openni nite game
```

3.2 Stub Tables

One of the first things added to `Critcl` after I took over maintenance was support for stubs-tables.

While Tcl and Tk provide such for portable linking and use of shared libraries only a few extensions actually do. There is no support for them in TEA and they need quite a lot of boilerplate in many places.

With `Critcl` supporting them directly through a few commands their use becomes much simpler. That said, a limitation of `Critcl`'s support is that it is a walled garden. Packages based on `Critcl` can consume the stubs generated by other `Critcl`-based packages, and export them to such. There is no cross-over with regular stubs however.

[1] In part hindered by the TEA not supporting its generation

`::critcl::api import` *name version*	Import stubs
`::critcl::api function` *resulttype name arguments*	Declare function exported through stubs
`::critcl::api header` ?*pattern*...?	Declare additional headers for the exported stubs
`::critcl::api extheader` ?*file*...?	Declare external headers for the exported stubs

Table 2: Stubs Import & Export Commands

Listing 2: Stubs Export

```
critcl::api header c/common.h
critcl::api header c/image_type.h
critcl::api header c/image.h
critcl::api header c/volume.h
critcl::api header c/buffer.h
critcl::api header c/rect.h
critcl::api header c/interpolate.h

critcl::api function {const crimp_imagetype*} crimp_imagetype_find {
    {const char*} name
}

critcl::api function void crimp_imagetype_def {
    {const crimp_imagetype*} imagetype
}

critcl::api function Tcl_Obj* crimp_new_imagetype_obj {
    {const crimp_imagetype*} imagetype
}

critcl::api function int crimp_get_imagetype_from_obj {
    Tcl_Interp*        interp
    Tcl_Obj*           imagetypeObj
    crimp_imagetype**  imagetype
}

(...)
```

Listing 3: Stubs Import

```
critcl::api import crimp::core 0.2
```

3.3 Optional & Variadic Arguments

The initial `cproc` command found in `Critcl` was quite simple. One of its limitations was that the user could only declare procedures which take a static number of arguments. The moment a variable number of arguments had to be processed `cproc` could be not be used anymore, and `ccommand` was required, putting the burden for the conversion of arguments and results back on the developer.

Since version 3.1.16 this limitation of `cproc` is fully fixed, enabling developers to declare procedures with optional arguments, and an unlimited number of arguments, with syntax similar to the Tcl core's builtin `proc`.

Listing 4: Optional cproc arguments

```
critcl::cproc optional_middle {int a int {b 1} int {c 2} int d} void {
    printf ("M|%d|%d|%d|%d|\n", a,b,c,d);
    fflush(stdout);
}
```

Regarding listing 5 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 5: Optional cproc arguments: Generated C code

```
#define ns__optional_middle10 "::optional_middle"

static void c__optional_middle10(int a, int has_b, int b, int has_c, int c, int d)
{
  printf ("M|%d|%d|%d|%d|\n", a,b,c,d);
  fflush(stdout);
}

static int
tcl__optional_middle10(ClientData cd, Tcl_Interp *interp, int oc, Tcl_Obj *CONST ov[])
{
  int _a;  int _has_b = 0;
  int _b;  int _has_c = 0;
  int _c;
  int _d;
  int idx_;
  int argc_;

  if ((oc < 3) || (5 < oc)) {
    Tcl_WrongNumArgs(interp, 1, ov, "a ?b? ?c? d");
    return TCL_ERROR;
  }

  /* (int a) - - -- --- ----- -------- */
  { if (Tcl_GetIntFromObj(interp, ov[1], &_a) != TCL_OK) return TCL_ERROR; }

  idx_  = 2;
  argc_ = oc - 2;

  /* (int b, optional, default 1) - - -- --- ----- -------- */
  if (argc_ > 1) {
    { if (Tcl_GetIntFromObj(interp, ov[idx_], &_b) != TCL_OK) return TCL_ERROR; }
    idx_++;
    argc_--;
    _has_b = 1;
  } else {
    _b = 1;
  }

  /* (int c, optional, default 2) - - -- --- ----- -------- */
  if (argc_ > 1) {
    { if (Tcl_GetIntFromObj(interp, ov[idx_], &_c) != TCL_OK) return TCL_ERROR; }
    idx_++;
    argc_--;
    _has_c = 1;
  } else {
    _c = 2;
  }

  /* (int d) - - -- --- ----- -------- */
  { if (Tcl_GetIntFromObj(interp, ov[idx_], &_d) != TCL_OK) return TCL_ERROR; }

  /* Call - - -- --- ----- -------- */
  c__optional_middle10(_a, _has_b, _b, _has_c, _c, _d);

  /* (void return) - - -- --- ----- -------- */
  return TCL_OK;
}
```

Listing 6: cproc args handling

```
critcl::cproc variadic {int args} void {
    int i;
    for (i=0; i < args.c; i++) printf ("[%2d] = %d\n", i, args.v[i]);
    fflush(stdout);
}
```

Regarding listing 7 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 7: cproc args handling: Generated C code

```
#define ns__variadic4 "::variadic"

#ifndef CRITCL_variadic_int
#define CRITCL_variadic_int

typedef struct critcl_variadic_int {
  int  c; /* Element count */
  int* v; /* Allocated array of the elements */
} critcl_variadic_int;

static int
_critcl_variadic_int_item (Tcl_Interp* interp, Tcl_Obj* src, int* dst) {
  { if (Tcl_GetIntFromObj(interp, src, dst) != TCL_OK) return TCL_ERROR; }
  return TCL_OK;
}

#endif /* CRITCL_variadic_int --------- */

static void c__variadic4(critcl_variadic_int args)
{
  int i;
  for (i=0; i < args.c; i++) printf ("[%2d] = %d\n", i, args.v[i]);
  fflush(stdout);
}

static int
tcl__variadic4(ClientData cd, Tcl_Interp *interp, int oc, Tcl_Obj *CONST ov[])
{
  critcl_variadic_int _args;
  /* (int args, ...) - - -- --- ----- --------- */
  {
    int src, dst, leftovers = (oc-1);
    _args.c = leftovers;
    _args.v = (int*) ((!leftovers) ? 0 : ckalloc (leftovers * sizeof (int)));
    for (src = 1, dst = 0; leftovers > 0; dst++, src++, leftovers--) {
      if (_critcl_variadic_int_item (interp, ov[src], &(_args.v[dst])) != TCL_OK) {
        ckfree ((char*) _args.v); /* Cleanup partial work */
        return TCL_ERROR;
      }
    } }

  /* Call - - -- --- ----- --------- */
  c__variadic4(_args);

  /* (Release: int args, ...) - - -- --- ----- --------- */
  if (_args.c) { ckfree ((char*) _args.v); }

  /* (void return) - - -- --- ----- --------- */
  return TCL_OK;
}
```

The handling of *args* is also a demonstration of the power of the support for custom types described in the next section, generating the necessary conversion from the conversion of the declared base-type.

Listing 8: cproc 'args' type generation

```
proc ::critcl::MakeVariadicTypeFor {type} {
    set ltype variadic_$type
    if {![has-argtype $ltype]} {
        lappend one @@  src
        lappend one &@A dst
        lappend one @A  *dst
        lappend one @A. dst->

        lappend map @1conv@ [Deline [string map $one [ArgumentConversion $type]]]
        lappend map @type@  [ArgumentCType $type]
        lappend map @ltype@ $ltype

        argtype $ltype [string map $map {
            int src, dst, leftovers = @C;
            @A.c = leftovers;
            @A.v = (@type@*) ((!leftovers) ? 0 : ckalloc (leftovers * sizeof (@type@)));
            for (src = @I, dst = 0; leftovers > 0; dst++, src++, leftovers--) {
                if (_critcl_variadic_@type@_item (interp, ov[src], &(@A.v[dst])) != TCL_OK) {
                    ckfree ((char*) @A.v); /* Cleanup partial work */
                    return TCL_ERROR;
                }
            }
        }] critcl_$ltype critcl_$ltype

        argtypesupport $ltype [string map $map {
            /* NOTE: Array 'v' is allocated on the heap. The argument
            // release code is used to free it after the worker
            // function returned. Depending on type and what is done
            // by the worker it may have to make copies of the data.
            */

            typedef struct critcl_@ltype@ {
                int     c; /* Element count */
                @type@* v; /* Allocated array of the elements */
            } critcl_@ltype@;

            static int
            _critcl_variadic_@type@_item (Tcl_Interp* interp, Tcl_Obj* src, @type@* dst) {
                @1conv@
                return TCL_OK;
            }
        }]

        argtyperelease $ltype [string map $map {
            if (@A.c) { ckfree ((char*) @A.v); }
        }]
    }
    return $ltype
}
```

3.4 Custom Types

Another problem of the initial `cproc` was its limited support for C types. While the chosen types were arguably the most important ones it became quickly a wall forcing developers back to the more burden-some `ccommand`.

Since version 3.1 this limitation is fixed, enabling developers to declare custom type(conversion)s, for both arguments and results. As part of this change the support for the existing types was also rewritten to use the new commands, substantially cleaning up the internals as well.

The previous section showed a complex example of the power of this feature already, where it was used to dynamically generate type(conversion)s for arrays of any already supported base-type. I should note that I have **not** attempted to create nested arrays, i.e. arrays of an array of some type.

`::critcl::has-resulttype` *name*	Test if a result-type is known
`::critcl::resulttype` *name body ?ctype?*	Declare a custom result conversion
`::critcl::resulttype` *name* `=` *origname*	Declare an alias for an existing conversion
`::critcl::has-argtype` *name*	Test if an argument-type is known
`::critcl::argtype` *name body ?ctype? ?ctypefun?*	Declare a custom argument-conversion
`::critcl::argtype` *name* `=` *origname*	Declare an alias for an existing conversion
`::critcl::argtypesupport` *name code*	Specify supporting code for conversion (structure definitions, and the like)
`::critcl::argtyperelease` *name script*	Release heap-allocated resources of an argument

Table 3: Type Definition Commands

Listing 9: Custom Argument Type

```
# kinetcl_pixelformat is defined in kt_image.tcl
critcl::argtype XnPixelFormat {
    if (Tcl_GetIndexFromObj (interp, @@,
                             kinetcl_pixelformat,
                             "pixelformat", 0, &@A) != TCL_OK) {
        return TCL_ERROR;
    }
    @A ++; /* Convert from Tcl's 0-indexed value to OpenNI's 1-indexing. */
} int int
```

Listing 10: Custom Result Type

```
critcl::resulttype XnPixelFormat {
    if (rv == (XnPixelFormat) -1) {
        Tcl_AppendResult (interp, "Inheritance error: Not an image generator", NULL);
        return TCL_ERROR;
    }
    /* ATTENTION: The array is 0-indexed, wheras the pixelformat 'rv' is 1-indexed */
    Tcl_SetObjResult (interp, Tcl_NewStringObj (kinetcl_pixelformat [rv-1],-1));
    return TCL_OK;
}
```

3.5 Diverting & Capturing Output

The standard behaviour for `Critcl` is to collect all the C code fragments in memory before assembling and writing them to a file when the time comes to compile everything. This collection is done on a per-file basis, keeping the information of different source files apart, except when explicitly asked for the opposite, see `critcl::source`.

On the other hand, the same foundation can be used to keep things apart which normally would go together, by using *virtual files*. They are organized as a stack and were introduced to support higher-level packages like the generators we will discuss in section 4. Their main purpose is to allow generators to intercept and capture the output of low-level critcl commands for their own purpose, like additional templating and other transformations.

An important user is the `critcl::class` package (Section 4.6). Class- and instance methods can be written as either `ccommand` and `cproc` equivalents, with the package internally simply delegating to the associated low-level commands and capturing their output to ensure its own proper organization of the final C code.

Another advantage of this behaviour, beyond the trivial of not having to code up a duplicate implementation of `cproc`'s, is that methods automatically inherit all features and extensions of the underlying commands. While this is not so important for `ccommand`'s, which have not changed at all since inception, the same cannot be said for `cproc`'s. Custom argument- and result-types, support for optional arguments, handling of *args*, all are supported by `critcl::class` without having to modify the package at all.

`::critcl::collect_begin`	Begin new level of capturing
`::critcl::collect_end`	End level and return captured code
`::critcl::collect` *script*	Run script and capture code

Table 4: Capturing Code

Listing 11: Use of diversion in critcl::class

```
proc ::critcl::class::MethodExplicit {name mtype arguments args} {
    # mtype in {proc, command}
    MethodCheck method instance $name

    set bloc     [critcl::at::get]
    set enum     [MethodEnum method $name]
    set function ${enum}_Cmd
    set cdimport "[critcl::at::here!]    @instancetype@ instance = (@instancetype@) clientdata;"

    if {$mtype eq "proc"} {
        # Force availability of the interp in methods.
        if {[lindex $arguments 0] ne "Tcl_Interp*"} {
            set arguments [linsert $arguments 0 Tcl_Interp* interp]
        }

        lassign $args rtype body

        set body   $bloc[string trimright $body]
        set cargs  [critcl::argnames $arguments]
        if {[llength $cargs]} { set cargs " $cargs" }
        set syntax "/* Syntax: <instance> $name$cargs */"
        set body   "\n    $syntax\n$cdimport\n    $body"

        set code [critcl::collect {
            critcl::cproc $function $arguments $rtype $body -cname 1 -pass-cdata 1 -arg-offset 1
        }]

(...)
```

3.6 Locating Issues

Nobody writes bug-free code. That makes it important to know where the issues are when the compiler reports them. `Critcl` handles this by emitting appropriate `#line` pragmas which tell the C compiler where in the Tcl sources each piece of C code can be found, since its inception.

However with the generator packages discussed in section 4 we get more layers on top, i.e. Tcl code generating Tcl code containing embedded (generated) C code, and so on. With the existing system this caused problems in user code to be reported relative to locations in the generator's Tcl code, and not the user's code.

To fix this a number of commands exposing the internal handling of locations was added. With them a generator package can now easily place `#line` pragmas in front of user code before handing it to the next lower level. As each level places their pragma in front of that it will be the pragma from the outermost level which is last seen by the C compiler and used for location reporting, as we want it.

`::critcl::at::caller`	Save the current location at the caller
`::critcl::at::caller` *offset*	As above, plus the line offset
`::critcl::at::caller` *offset level*	As above, with the base location taken from a different stack level
`::critcl::at::here`	Save current location in current procedure
`::critcl::at::get*`	Return stored location as `#line` pragma
`::critcl::at::get`	As above, and clears the store
`::critcl::at::=` *file line*	Explicitly set the stored location
`::critcl::at::incr` *n...*	Modify the stored location
`::critcl::at::incrt` *str...*	As above, counting lines in the strings
`::critcl::at::caller!`	Combine `caller` and `get`
`::critcl::at::caller!` *offset*	Ditto
`::critcl::at::caller!` *offset level*	Ditto
`::critcl::at::here!`	Combine `here` and `get`

Table 5: Location Support Commands

Listing 12: Location command usage

```
proc ::critcl::iassoc::def {name arguments struct constructor destructor} {
    critcl::at::caller
    critcl::at::incrt $arguments   ; set sloc [critcl::at::get*]
    critcl::at::incrt $struct      ; set cloc [critcl::at::get*]
    critcl::at::incrt $constructor ; set dloc [critcl::at::get]

    set struct      $sloc$struct
    set constructor $cloc$constructor
    set destructor  $dloc$destructor

    (...)
    lappend map @struct@      $struct
    lappend map @constructor@ $constructor
    lappend map @destructor@  $destructor
    (...)

    critcl::util::Put $header [string map $map $template]
    critcl::ccode "#include <$hdr>"
    return
}
```

4. SUPPORT PACKAGES

4.1 Ekekos

One important pattern for the creation of thread-oblivious extensions to Tcl is to place the "global" state of the extension into a structure and then create and attach an instance of that structure to any interpreter loading that extension, via `Tcl_SetAssocData` [7] and related APIs.

The not so nice part about this pattern is that very often more than 50% of the code needed to be written is just boilerplate, first ensuring that the structure is initialized only once, and second that it is properly finalized when the interpreter it is attached to gets destroyed. In the chosen example we have 7 lines of user code embedded in 25 lines of boilerplate, more than 3/4 of the total code (32 lines).

The support package `critcl::iassoc` [25] was written to take on the burden of creating all that boilerplate. All it needs are the definition of the structure, and the C code fragments for initialization and finalization. Everything else will be generated around that. The generated C-level API consists of a single function to retrieve and initialize (once) the structure.

All the other supporting packages described in the following sections, with the exception of the general utilities, make use of this generator to handle the "global" state of their C code.

Listing 13: Ekeko: Declaration

```
critcl::iassoc::def marpatcl_context {} {
    Marpa_Config      config;
    Marpa_Grammar     grammar;     /* Communication: Grammar -> Recognizer constructor */
    Marpa_Recognizer  recognizer;  /* Communication: Recognizer -> Bocage constructor */
} {
    data->grammar    = NULL;
    data->recognizer = NULL;
    (void) marpa_c_init (&data->config);
} {
    /* nothing to do */
}
```

Regarding listing 14 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 14: Ekeko: Generated C code

```c
typedef struct marpatcl_context_data__ {
    Marpa_Config       config;
    Marpa_Grammar      grammar;     /* Communication: Grammar -> Recognizer constructor */
    Marpa_Recognizer   recognizer;  /* Communication: Recognizer -> Bocage constructor */
} marpatcl_context_data__;

typedef struct marpatcl_context_data__* marpatcl_context_data;

static void
_marpa62_iassoc_marpatcl_context_Release (marpatcl_context_data data, Tcl_Interp* interp)
{
    /* nothing to do */
    ckfree((char*) data);
}

static marpatcl_context_data
_marpa62_iassoc_marpatcl_context_Init (Tcl_Interp* interp)
{
    marpatcl_context_data data = (marpatcl_context_data) ckalloc (sizeof (marpatcl_context_data__));

    data->grammar    = NULL;
    data->recognizer = NULL;
    (void) marpa_c_init (&data->config);
    return data;

  error:
    ckfree ((char*) data);
    return NULL;
}

static marpatcl_context_data
marpatcl_context (Tcl_Interp* interp)
{
#define KEY "critcl::iassoc/p=marpa/a=marpatcl_context"

    Tcl_InterpDeleteProc* proc = (Tcl_InterpDeleteProc*) _marpa62_iassoc_marpatcl_context_Release;
    marpatcl_context_data data;

    data = Tcl_GetAssocData (interp, KEY, &proc);
    if (data) {
        return data;
    }

    data = _marpa62_iassoc_marpatcl_context_Init (interp);

    if (data) {
        Tcl_SetAssocData (interp, KEY, proc, (ClientData) data);
    }

    return data;
#undef KEY
}
```

4.2 String Pools

Many packages will have a fixed and small set of string constants occuring in a few places. Most of these will be coded to simply create a new string *Tcl_Obj** from a const *char** every time the constant is needed, as this is easy to do, despite the inherent waste of memory. There is otherwise just too much boilerplate involved, especially when the extension is to be thread-safe.

The support package `critcl::literals` [28] was written to tilt things the other way, to make the declaration and management of string pools which do not waste memory as easy as the normal solution, hiding all attendant complexity from the user.

Most of the boilerplate is actually handled by `critcl::iassoc` (Section 4.1), with `critcl::literals` itself just a thin wrapper which adds all the pool-specific code. The generated C-level API consists of a function converting from integer to string, an enumeration, and a header file. The function is further registered as a `cproc` result type.

Listing 15: Literal pool: Declaration

```
critcl::literals::def marpatcl_step {
    mt_s_rule       "rule"
    mt_s_token      "token"
    mt_s_nulling    "null"
    mt_s_0          "first"
    mt_s_n          "last"
    mt_s_id         "id"
    mt_s_res        "dst"
    mt_s_value      "value"
    mt_s_end_es     "end-es"
    mt_s_start_es   "start-es"
}
```

Regarding listing 16 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 16: Literal pool: Generated C code

```
typedef struct marpatcl_step_iassoc_data__ {
        /* Array of the string literals, indexed by the symbolic names */
        Tcl_Obj* literal [marpatcl_step_name_LAST];
} marpatcl_step_iassoc_data__;

static void
_marpa40_iassoc_marpatcl_step_iassoc_Release (marpatcl_step_iassoc_data data, Tcl_Interp* interp)
{
            Tcl_DecrRefCount (data->literal [mt_s_rule]);
(...)
}

static marpatcl_step_iassoc_data
_marpa40_iassoc_marpatcl_step_iassoc_Init (Tcl_Interp* interp)
{
    marpatcl_step_iassoc_data data = (marpatcl_step_iassoc_data) ckalloc (sizeof (marpatcl_step_iasso

            data->literal [mt_s_rule] = Tcl_NewStringObj ("rule", -1);
            Tcl_IncrRefCount (data->literal [mt_s_rule]);
            (...)
}

(...)

Tcl_Obj*
marpatcl_step (Tcl_Interp*  interp,
        marpatcl_step_names literal)
{
    if ((literal < 0) || (literal >= marpatcl_step_name_LAST)) {
        Tcl_Panic ("Bad marpatcl_step literal");
    }
    return marpatcl_step_iassoc (interp)->literal [literal];
}
```

4.3 Enumerations

A logical extension of the literal pools shown in the previous section are enumerations. Whereas a literal pool only allows the conversion of a C identifier to a Tcl string, an enumeration can be converted the other way as well, from Tcl string to C identifier.

This is what the support package `critcl::enum` [24] provides, the easy declaration of enumerations with representations at both C- and Tcl-level, which can be converted into each other.

Most of the needed boilerplate is actually handled by `critcl::iassoc` (Section 4.1), with `critcl::enum` itself just a thin wrapper which adds all the enumeration-specific code. The generated C-level API consists of an enumeration, two conversion functions (integer to string and vice versa), and a header file. The two functions are further registered as `cproc` argument and result types, respectively.

Note that package **defines** the underlying C-level *enum* type. This means that this package is not useful for writing bindings to existing enumerations provided by external libraries. To do that use the enum- and bit-map packages instead. They are explained in the following sections, 4.4 and 4.5.

Listing 17: Enumeration: Declaration

```
critcl::enum::def demo {
    E_global  global
    E_exact   exact
    E_filler  filler
}
```

Regarding listing 18 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 18: Enumeration: Generated C code

```
typedef struct demo_pool_iassoc_data__ {
        /* Array of the string literals, indexed by the symbolic names */
        Tcl_Obj* literal [demo_pool_name_LAST];
} demo_pool_iassoc_data__;
typedef struct demo_pool_iassoc_data__* demo_pool_iassoc_data;

static void
_enum6_iassoc_demo_pool_iassoc_Release (demo_pool_iassoc_data data, Tcl_Interp* interp)
{
            Tcl_DecrRefCount (data->literal [E_global]);
            (...)
}

static demo_pool_iassoc_data
_enum6_iassoc_demo_pool_iassoc_Init (Tcl_Interp* interp)
{
    demo_pool_iassoc_data data = (demo_pool_iassoc_data) ckalloc (sizeof (demo_pool_iassoc_data__));

            data->literal [E_global] = Tcl_NewStringObj ("global", -1);
            Tcl_IncrRefCount (data->literal [E_global]);
            (...)
}
(...)

typedef enum demo_pool_names {
  E_global, E_exact, E_filler, demo_pool_name_LAST
} demo_pool_names;

#define demo_ToObj(i,l) (demo_pool(i,l))

extern int
demo_GetFromObj (Tcl_Interp* interp, Tcl_Obj* obj, int flags, int* literal )
{
  static const char* strings[4] = { "global", "exact", "filler", NULL };
  return Tcl_GetIndexFromObj (interp, obj, strings, "demo", flags, literal);
}
```

4.4 Enum Maps

The supporting package `critcl::emap` [23] is a variant of `critcl::enum`. It was written to support the case where the C enumeration (or equivalent) to map to Tcl is provided externally. The expected use-case is writing bindings for some other library.

The generated C-level API consists of two conversion functions (integer to string and vice versa), and a header file. The two functions are further registered as `cproc` argument and result types.

Listing 19: Enumeration mapping: Declaration

```
critcl::emap::def marpatcl_steptype {
    step-rule       MARPA_STEP_RULE
    step-token      MARPA_STEP_TOKEN
    step-nulling    MARPA_STEP_NULLING_SYMBOL
    step-inactive   MARPA_STEP_INACTIVE
    step-initial    MARPA_STEP_INITIAL
    step-internal1  MARPA_STEP_INTERNAL1
    step-internal2  MARPA_STEP_INTERNAL2
    step-trace      MARPA_STEP_TRACE
}
```

Regarding listing 20 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 20: Enumeration mapping: Generated C code

```
typedef struct marpatcl_steptype_iassoc_data__ {
        const char*     c      [8+1]; /* State name, C string */
        Tcl_Obj*        tcl    [8];   /* State name, Tcl_Obj*, sharable */
        int             value  [8];   /* State code */
} marpatcl_steptype_iassoc_data__;
typedef struct marpatcl_steptype_iassoc_data__* marpatcl_steptype_iassoc_data;

static void
_marpa48_iassoc_marpatcl_steptype_iassoc_Release (marpatcl_steptype_iassoc_data data, Tcl_Interp* int
{
            Tcl_DecrRefCount (data->tcl [5]);
(...)
}

static marpatcl_steptype_iassoc_data
_marpa48_iassoc_marpatcl_steptype_iassoc_Init (Tcl_Interp* interp)
{
    marpatcl_steptype_iassoc_data data = (marpatcl_steptype_iassoc_data) ckalloc (sizeof (marpatcl_st

            data->c      [5] = "step-rule";
            data->value  [5] = MARPA_STEP_RULE;
            data->tcl    [5] = Tcl_NewStringObj ("step-rule", -1);
            Tcl_IncrRefCount (data->tcl [5]);
(...)
}
(...)
int
marpatcl_steptype_encode (Tcl_Interp* interp, Tcl_Obj* state, int* result)
{
  marpatcl_steptype_iassoc_data context = marpatcl_steptype_iassoc (interp);
  int id, res = Tcl_GetIndexFromObj (interp, state, context->c, "marpatcl_steptype", 0, &id);
  if (res != TCL_OK) {
    Tcl_SetErrorCode (interp, "MARPATCL_STEPTYPE", "STATE", NULL);
    return TCL_ERROR;
  }
  *result = context->value [id];
  return TCL_OK;
}
```

```
Tcl_Obj*
marpatcl_steptype_decode (Tcl_Interp* interp, int state)
{
  char buf [20];
  int i;
  marpatcl_steptype_iassoc_data context = marpatcl_steptype_iassoc (interp);

  for (i = 0; i < 8; i++) {
    if (context->value[i] != state) continue;
    return context->tcl [i];
  }
  sprintf (buf, "%d", state);
  Tcl_AppendResult (interp, "Invalid marpatcl_steptype state code ", buf, NULL);
  Tcl_SetErrorCode (interp, "MARPATCL_STEPTYPE", "STATE", NULL);
  return NULL;
}
```

4.5 Bitmaps & Flags

The supporting package `critcl::bitmap` [19] is an outgrowth[2] of `critcl::emap`. Its use-case is the conversion of bit-sets instead of individual integers, with the flags in the set described by an enumeration (or equivalent). At the Tcl-level such sets are represented as lists of strings.

A unique feature of the package is the optional exclusion list. This feature was added to support the declaration of flags for which only encoding makes sense, but not decoding. The expected use-case are flag values which represent a combination of other flags in the mapped enumeration.

The generated C-level API consists of two conversion functions (integer (bitset) to list of strings and vice versa), and a header file. The two functions are further registered as `cproc` argument and result types, respectively.

Listing 21: Bitset mapping: Declaration

```
critcl::include sys/inotify.h

critcl::bitmap::def tcl_inotify_events {
    accessed        IN_ACCESS
    all             IN_ALL_EVENTS
    attribute       IN_ATTRIB
    closed          IN_CLOSE
    closed-nowrite  IN_CLOSE_NOWRITE
    closed-write    IN_CLOSE_WRITE
    created         IN_CREATE
    deleted         IN_DELETE
    deleted-self    IN_DELETE_SELF
    dir-only        IN_ONLYDIR
    dont-follow     IN_DONT_FOLLOW
    modified        IN_MODIFY
    move            IN_MOVE
    moved-from      IN_MOVED_FROM
    moved-self      IN_MOVE_SELF
    moved-to        IN_MOVED_TO
    oneshot         IN_ONESHOT
    open            IN_OPEN
    overflow        IN_Q_OVERFLOW
    unmount         IN_UNMOUNT
} { all closed move oneshot }
```

[2] Historically speaking it actually existed before `critcl::emap`

Regarding listing 20 please note that the shown code is not exactly as generated. It was modified to better fit the pages, by removing `#line` pragmas, comments, and other irrelevant lines.

Listing 22: Bitset mapping: Generated C code

```
typedef struct tcl_inotify_events_iassoc_data__ {
            const char*     c     [20+1]; /* Bit name, C string */
            Tcl_Obj*        tcl   [20];   /* Bit name, Tcl_Obj*, sharable */
            int             mask  [20];   /* Bit mask */
            int             recv  [20];   /* Flag, true for receivable event */
} tcl_inotify_events_iassoc_data__;
typedef struct tcl_inotify_events_iassoc_data__* tcl_inotify_events_iassoc_data;

static void
_inotify34_iassoc_tcl_inotify_events_iassoc_Release (tcl_inotify_events_iassoc_data data,
                                                     Tcl_Interp* interp)
{
            Tcl_DecrRefCount (data->tcl [0]);
            (...)
}

static tcl_inotify_events_iassoc_data
_inotify34_iassoc_tcl_inotify_events_iassoc_Init (Tcl_Interp* interp)
{
    tcl_inotify_events_iassoc_data data =
      (tcl_inotify_events_iassoc_data) ckalloc (sizeof (tcl_inotify_events_iassoc_data__));
                data->c     [0] = "accessed";
                data->mask  [0] = IN_ACCESS;
                data->recv  [0] = 1;
                data->tcl   [0] = Tcl_NewStringObj ("accessed", -1);
                Tcl_IncrRefCount (data->tcl [0]);
                (...)
}
(...)
int
tcl_inotify_events_encode (Tcl_Interp* interp, Tcl_Obj* flags, int* result)
{
  tcl_inotify_events_iassoc_data context = tcl_inotify_events_iassoc (interp);
  int mask, lc, i, id;
  Tcl_Obj** lv;
  if (Tcl_ListObjGetElements (interp, flags, &lc, &lv) != TCL_OK) {
    return TCL_ERROR;
  }
  mask = 0;
  for (i = 0; i < lc; i++)  {
    if (Tcl_GetIndexFromObj (interp, lv[i], context->c, "tcl_inotify_events", 0,
                             &id) != TCL_OK) {
      Tcl_SetErrorCode (interp, "TCL_INOTIFY_EVENTS", "FLAG", NULL);
      return TCL_ERROR;
    }
    mask |= context->mask [id];
  }
  *result = mask;
  return TCL_OK;
}

Tcl_Obj*
tcl_inotify_events_decode (Tcl_Interp* interp, int mask)
{
  int i;
  tcl_inotify_events_iassoc_data context = tcl_inotify_events_iassoc (interp);
  Tcl_Obj*                        res     = Tcl_NewListObj (0, NULL);
```

```
    for (i = 0; i < 20; i++) {
        if (!context->recv[i])          continue;
        if (!(mask & context->mask[i])) continue;
        (void) Tcl_ListObjAppendElement (interp, res, context->tcl [i]);
    }
    return res;
}
```

4.6 Classes & Objects

Writing classes in Tcl is simple[3]. Writing classes in C is not that complicated either. Writing many tens of classes, well now the boilerplate for setting up the global state, the class structures, dispatch, etc. becomes tedious.

That was the situation I faced when I took on the `KineTcl` project. The external library to bind to, `OpenNI` [9] (v1) was very object-oriented, providing just shy of 20 classes.

From this the supporting package `critcl::class` [20] was born. And a few other things already mentioned in preceding sections (Diversions, Custom types, Ekekos). While I still had to write the methods themselves[5], everything else was generated.

An important early decision was to reuse the existing parts of critcl as much as possible. I.e. allow class- and instance methods to be the equivalent of either `ccommand` or `cproc`, and delegate the main handling of the user's code to these commands. This has paid off since then, with all the extensions of `cproc` automatically available to classes without any additional effort.

Normally I would now include an example of a class here, as was done for the preceding packages, followed by the C code generated from it. Unfortunately the C code for classes is usually so large, it will not really fit, even with editing. As such I recommend to go and take a look instead at either the examples in `Critcl`, or the `TclYAML` and `Marpa` packages. While `KineTcl` is where it started, the additional higher-level generation of classes it does on top of the basic OO support tends to muddy the waters, making it a bad introductory example.

4.7 General Utilities

The supporting package `critcl::util` [29] is, unlike all preceding packages, not a generator.

It provides a bare-bones set of utility commands to check the build environment and record the result of such checks.

`::critcl::util::locate` *label paths ?cmd?*	Search a file in a list of directories, possibly filtered
`::critcl::util::checkfun` *name ?label?*	Test ability to link function
`::critcl::util::def` *path define ?value?*	Add a `#define` to a config file
`::critcl::util::undef` *path define*	Add an `#undef` to a config file

Table 6: Build environment introspection

Listing 23: `CRIMP` environment checks

```
if {[critcl::util::checkfun lrint]} {
    critcl::msg -nonewline "(native lrint()) "
} else {
    critcl::msg -nonewline "(+ compat/lrint.c) "
    critcl::csources compat/lrint.c
}
::apply {{} {
    foreach f { hypotf sinf cosf sqrtf expf logf atan2f } {
        set fd [string range $f 0 end-1]
        set d  C_HAVE_[string toupper $f]
        if {[critcl::util::checkfun $f]} {
            critcl::util::def crimp_config.h $d
            critcl::msg -nonewline "(have $f) "
        } else {
            critcl::util::undef crimp_config.h $d
            critcl::msg -nonewline "($f -> $fd) "
        }
    }
}}
```

[3]The problem is choosing among the multitude of available OO packages.[4]
[4]Ok, these days I pretty much use only `TclOO`
[5]Several important parts were even amenable to higher-level generation, providing another boost.

5. INCOMPATIBILITIES

Since Steve Landers's paper[10] in 2002, which described version 2.0 we have moved to version 3.x, a major version change which introduced two incompatibilities (or vice versa). These are

1. The command `critcl::platform` was deprecated in version 2.1, and removed in version 3.0. It is superceded by `critcl::targetplatform`.
2. The command `critcl::compiled` was kept with in version 2.1 with semantics in contradiction to its documentation and name. This contradiction was removed in version 3.0, with the visible semantics of the command changed to be in line with its name.

6. MUSINGS

Regarding future development I am currently pondering

1. Extending `critcl::emap` with an option signaling more knowledge about the enumeration to map, like "Ordered", "no gaps between values", "values starting at 0 (or fixed n)", etc. All properties which can allow the package to generate more efficient code.
2. Modifying `critcl::bitmap` so that its decoder is able to decode multi-flag combinations.
3. Create a variant of `critcl::literals` managing a dynamic pool of strings, for use with packages where user-provided strings may occur multiple times. I.e. a cache of common user-input we could share.
4. Support packages encapsulating common C code, like convenience macros for memory allocation, assertions, and tracing.

APPENDIX

A. REFERENCES

[1] `BAWT`, Paul Obermeier [13], `http://www.bawt.tcl3d.org/`
[2] `kbskit`, Rene Zaumseil [13], `https://sourceforge.net/projects/kbskit/`
[3] `Kettle`, Andreas Kupries [11], `https://core.tcl.tk/akupries/kettle`
[4] `Quill`, Will Duquette [16], `https://github.com/wduquette/tcl-quill`
[5] ActiveState, `https://www.activestate.com`
[6] `https://core.tcl.tk`, Various
[7] `https://www.tcl.tk/man/tcl8.6/TclLib/AssocData.htm`
[8] `https://www.tcl.tk/man/tcl8.6/TclLib/GetIndex.htm`
[9] `OpenNI`, `https://en.wikipedia.org/wiki/OpenNI`
[10] "Critcl - Beyond Stubs and Compilers", Steve Landers [15],
PDF: `http://www.digitalsmarties.com/Tcl2002/critcl.pdf`,
Slides: `http://equi4.com/papers/ctpaper1.html`
[11] Andreas Kupries, `https://core.tcl.tk/akupries/projects.html`
[12] Jean-Claude Wippler, `https://github.com/jcw/`
[13] Paul Obermeier, `http://www.posoft.de/`
[14] Rene Zaumseil, `https://wiki.tcl.tk/18145`
[15] Steve Landers, `https://www.digitalsmarties.com/`
[16] Will Duquette, `http://www.wjduquette.com/`
[17] Sean Wood, `http://www.etoyoc.com/yoda/`
[18] Ekeko, `https://wiki.tcl.tk/ekeko`
[19] `critcl::bitmap` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_bitmap.html`
[20] `critcl::class` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_class.html`
[21] `CRIMP`, Andreas Kupries [11], `http://core.tcl.tk/akupries/crimp`
[22] `Critcl`, Andreas Kupries [11], Steve Landers [15], Jean-Claude Wippler [12],
`http://andreas-kupries.github.io/critcl/`
[23] `critcl::emap` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_emap.html`
[24] `critcl::enum` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_enum.html`
[25] `critcl::iassoc` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_iassoc.html`
[26] `KineTcl`, Andreas Kupries [11], `https://core.tcl.tk/akupries/kinetcl`
[27] `TclLinenoise`, Andreas Kupries [11], `https://github.com/andreas-kupries/tcl-linenoise`
[28] `critcl::literals` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_literals.html`
[29] `critcl::util` documentation, `http://andreas-kupries.github.io/critcl/doc/files/critcl_util.html`
[30] `Marpa`, Andreas Kupries [11], `https://core.tcl.tk/akupries/marpa`
[31] `PracTcl`, Sean Wood [17], `http://wiki.tcl.tk/42543`
[32] `TclYAML`, Andreas Kupries [11], `https://core.tcl.tk/akupries/tclyaml`