DylanWorks Foreign Function Interface

Roger Jarrett & Jonathan Bachrach

1. Introduction

The purpose of this paper is to describe the low-level Foreign Function Interface (FFI). The purpose of the FFI is to allow a Dylan component to call (or be called by) a component written in another language. We are particularly interested in interfacing with C and C++ on the Windows 95™ operating system. This includes libraries written in any language that have a C or C++ interface.

2. Basic Design Goals

The basic design goals for the low-level FFI are:

- Provide interoperability with other components on equal terms, so that users can build seamless, hybrid applications.
- Provide the substrate for the Creole header-mapping facilities.
- Provide a mapping between Dylan and C/C++ function parameters.
- Provide a mapping between Dylan and C/C++ function return values.
- Provide a mapping between Dylan and C/C++ constants and data.
- Interact with the GC in a safe (and sane) fashion.

3. Issues Regarding Dylan and C/C++

Translate Dylan data types to C datatypes following the conventions required by the native compiler. C passes parameters by value; however; that value may be a pointer to memory that could be modified by the callee. Special care must taken when passing some C data types as arguments to functions. These data types are basically any non-integer-like quantity; this includes pointers if the following holds:

```
sizeof(char *) != sizeof(int)
```

C parameter types that require special care are:

- `char` and `short` (passed as a right or left justified `int`, depending on the architecture).
- `double` (in floating point register or on stack; also watch the alignment of doubles).
- `float` (normally promoted to be a double unless using ANSI-C with prototypes).
- Structures (every architecture and compiler does this differently).

There are also some function return types that require special care:

- `char` and `short`.
- `double` and `float`.
- Structures.

3.1 ANSI-C versus K&R C

The ANSI-C and K&R C are two different dialects of C; the ANSI-C standard defines many of the areas of the language that were previously left as “implementor defined”.

One area of improvement of ANSI-C is the advent of function prototypes. The presence of ANSI-C prototypes in code changes the way that parameters are passed to functions. With function prototypes one can determine the ordered set of arguments to a function as well as its return type.

ANSI-C also provides the `const` keyword which could assist in type mapping and or error checking of the low-level FFI interface.

Therefore in the low-level FFI, we need to know the dialect of C we are calling.

3.2 C versus C++

Creole will handle the name mapping of the Dylan name to the C++ external name.

- Other C++ issues **<to be filled in>**

3.3 Error Checking

We could provide two versions of the FFI., and debugging and a production version. The debugging version would check for common errors including:

- Incorrect memory management.
- Checking that `const` types are constant.
- Checking that variables using a user specified expression.
- GC consistency checks before/after FFI.
- “Unusual” flow control, such as `longjmp`’s and signalling (i.e., C++ exceptions).

4. Basic Strategy

The basic strategy is to map “alien” types into the Dylan type system such that different types of C pointers have distinguishable classes when imported into Dylan. (Note that this is different from the strategy presently used in LispWorks). One advantage of doing this is that per-type converter methods can be defined to control the import/export mapping.

There will also be conversion hooks allowing users to control how an alien value gets mapped to a Dylan value, and vice versa.

The alien type system is based on a number of "generic" types ("generic" in the sense that they are not meant to be language specific). These types include:

- `<%alien-type> (<type>)`
- `<%alien-statically-typed-pointer> (<%alien-type>)`
- `<%alien-value-type> (<%alien-type>)`
- `<%alien-0-bit-value> (<%alien-value-type>)`
- `<%alien-8-bit-unsigned-value> (<%alien-value-type>)`
- `<%alien-8-bit-signed-value> (<%alien-value-type>)`
- `<%alien-16-bit-unsigned-value> (<%alien-value-type>)`
- `<%alien-16-bit-signed-value> (<%alien-value-type>)`
- `<%alien-32-bit-unsigned-value> (<%alien-value-type>)`
- `<%alien-32-bit-signed-value> (<%alien-value-type>)`
- `<%alien-64-bit-unsigned-value> (<%alien-value-type>)`
- `<%alien-64-bit-signed-value> (<%alien-value-type>)`
- `<%alien-ieee-single-float-value> (<%alien-value-type>)`
- `<%alien-ieee-double-float-value> (<%alien-value-type>)`
- `<%alien-ieee-extended-float-value> (<%alien-value-type>)`
- `<%alien-enumeration-value> (<%alien-value-type>)`
- `<%alien-aggregate-type> (<%alien-value-type>)`
- `<%alien-structure> (<%alien-aggregate-type>)`
- `<%alien-union> (<%alien-aggregate-type>)`
- `<%alien-array> (<%alien-aggregate-type>)`
- `<%alien-string> (<%alien-array>)`

The DylanWorks FFI library defines several macros that are used to define alien functions and values. They permit the specification of name and type mapping between Dylan and the Alien language. They are language specific, but for the C FFI they are:

- **define c-type**, which is used to define a new C type in terms of an existing C type of one of the primitive, generic types.
- **define c-variable**, which imports a C variable by specifying its C type and what Dylan type it should map to.
- **define c-constant**, which imports a C constant by specifying its initial value.
- **define c-struct**, which is used to define a new C structure data structure.

- **define c-union**, which is used to define a new C union data structure.
- **define c-function**, which imports an alien function by defining its contract (that is, its set of input parameters and their types, and its output parameters and their types).
- **export-as-c-function**, which makes a Dylan function callable from non-Dylan code.

Creole will use these building blocks to create the Dylan types determined during header file parsing.

5. C FFI Specification

This section describes the details of the C FFI. We first show how to specify C types and describe the syntax and mechanics of name and type mapping. We then proceed to describe the various FFI macros used to import and export variables and functions to and from Dylan.

5.1 C Type Specifiers

The DylanWorks FFI library provides a type language for defining new restricted types and for creating anonymous combinations of the above types. The purpose of this is to express C types that can not be readily expressed in Dylan and to allow the user to define types that can be inlined into aggregate data structures.

c-type-specifier:

```
c-type-name
c-bit-field-of(INTEGER)
c-signed-bit-field-of(INTEGER)
c-enum(SYMBOL ...)
c-pointer-to(c-type-specifier)
c-struct dylan->c-slot-specifier ... end
c-union dylan->c-slot-specifier ... end
c-array-of(c-type-specifier,
          #key dimensions = #(#F),
               layout = #"row-major")
c-function (dylan->c-parameter-specifier ...) =>
  c->dylan-parameter-specifier
```

c-type-name:

SYMBOL

dylan->c-slot-specifier:

```
slot SYMBOL :: dylan->c-type-specifier;
```

The following C struct

```
struct {
  char *name;
  long flags;
  union {
   long int_val;
   char *string_val;
  } val;
};
```

can be defined in Dylan as

```
c-struct
  slot name :: <C/string>;
  slot flags :: <C/long>;
  slot val ::
    c-union
      slot int_val :: <C/int>;
      slot string_val :: <C/string>;
    end;
end;
```

One important concept is the notion of type mapping between Dylan and C. The user is allowed to specify how types are mapped at the point of specifying the type (i.e., ***dylan->c-type-specifier***). The user may specify a low-level mapping (set off with `=>`, `<=`, or `<=>`)) and optionally a high-level mapping (set off with `->`, `<-`, or `<->`). Low-level mappings operate through simple coercions, while high-level mapping invoke the user defined functions `export-as` and `import-as` depending on the direction of the mapping.

much more explanation is needed here along with many more examples

dylan->c-type-specifier:

c-type-specifier
dylan-type-specifier `=>` ***c-type-specifier***
dylan-type-specifier `->` ***dylan-type-specifier*** `=>` ***c-type-specifier***
dylan-type-specifier `->` `(`***dylan-type-specifier, mapper)*** `=>` ***c-type-specifier***

dylan-type-specifier:

expression

c->dylan-type-specifier:

c-type-specifier
c-type-specifier `=>` ***dylan-type-specifier***
c-type-specifier `=>` ***dylan-type-specifier*** `->` ***dylan-type-specifier***
c-type-specifier `=>` ***dylan-type-specifier*** `->` `(`***dylan-type-specifier, mapper)***

Parameter specifications are treated specially because a parameter can not only be used to pass information into a function but also to pass information out of a function when the argument is passed call-by-reference.

more explanation required and lots more examples

> ***dylan->c-parameter-specifier:***
>
> > ***dylan->c-type-specifier***
> > **SYMBOL** `::` ***dylan->c-type-specifier***
> > ***output-dylan->c-parameter-specifier***
> > **SYMBOL** `::` ***output-dylan->c-parameter-specifier***
> > ***input-output-dylan->c-parameter-specifier***
> > **SYMBOL** `::` ***input-output-dylan->c-parameter-specifier***
>
> ***output-dylan->c-parameter-specifier:***
>
> > `<=` ***c-type-specifier***
> > ***dylan-type-specifier*** `<=` ***dylan-type-specifier***
> > ***dylan-type-specifier*** `<-` ***dylan-type-specifier*** `<=` ***c-type-specifier***
> > ***dylan-type-specifier*** `<-` `(`***dylan-type-specifier, mapper)*** `<=` ***c-type-specifier***
>
> ***input-output-dylan->c-parameter-specifier:***
>
> > `<=>` ***c-type-specifier***
> > ***dylan-type-specifier*** `<=>` ***dylan-type-specifier***
> > ***dylan-type-specifier*** `<->` ***dylan-type-specifier*** `<=>` ***c-type-specifier***
> > ***dylan-type-specifier*** `<->`
> > `(`***dylan-type-specifier, input-mapper, output-mapper)*** `<=>`
> > ***c-type-specifier***

5.2 Defining New C Types

It is possible to define type aliases for previously defined C types as follows.

> `define c-type` **SYMBOL** `=` ***c-type-specifier***

It is also possible to define new C types through subclassing. Slots may also be defined on these types.

> `define c-type` **SYMBOL** `(`***type-specifier ...***`)` ***slot-descriptor ...*** `end;`

We assume that the following is predefined:

```
define class <C/type> (<%alien-type>) end;
```

more explanation about how this is mixed in is in order

The DylanWorks FFI library defines some C / platform specific aliases such as C integers and pointers, and so on. The following are a representative set.

```
define c-type <C/statically-typed-pointer>
    (<%alien-statically-typed-pointer>)
end;
define c-type <C/short> (<%alien-16-bit-signed-value>) end;
define c-type <C/int> (<%alien-32-bit-signed-value>) end;
define c-type <C/long> (<%alien-64-bit-signed-value>) end;
define c-type <C/single> (<%alien-ieee-single-float-value>) end;
define c-type <C/double> (<%alien-ieee-double-float-value>) end;
define c-type <C/extended> (<%alien-ieee-extended-float-value>) end;
define c-type <C/string> (<%alien-string-value>, <string>) end;
define c-type <C/void> (<%alien-0-bit-value>) end;
define c-type <C/void*> = pointer-to(<C/void>);
```

Users may define their own types as well.

```
define c-type <X/boolean> = <C/int>;
define c-type <X/xt-position> = <C/short>;
define c-type <X/xt-dimension> = <C/short>;
define c-type <X/arg*> = pointer-to(<X/arg>);
```

5.3 Defining C Structures

The user may defined named C structs as follows.

`define c-struct` **SYMBOL** ***dylan->c-slot-specifier ...*** `end`

The following is a simple example.

```
define c-struct <X/arg>
  slot arg->name :: <X/string>;
  slot arg->value :: <X/xt-arg-val>;
end;
```

5.4 Defining C Unions

The user may defined named C unions as follows.

`define c-union` **SYMBOL** ***dylan->c-slot-specifier ...*** `end`

The following is a simple example.

```
define c-union <Float-Decoder>
  slot as-float :: <C/single>;
  slot as-bits  ::
    c-struct
      slot sign-bit :: c-bit-field-of(1);
      slot exponent :: c-bit-field-of(8);
      slot mantissa :: c-bit-field-of(23);
    end;
end;
```

5.5 Defining C Variables

C variables may be imported by defining them as follows.

`define c-variable` ***dylan->c-name-specifier*** `::` ***c->dylan-type-specifier***

dylan->c-name-specifier:

SYMBOL `=>` **STRING**

The ***dylan->c-name-specifier*** specifies the mapping from a Dylan name to a C name. The following is an example of its usage

```
define c-variable number-of-processes => "NumberOfProcesses" ::
  <C/int> => <integer>;
```

The system automatically defines a getter and setter for the C variable.

```
number-of-processes()
number-of-processes() := 57;
```

5.6 Defining C Constants

C constants may be imported by defining them as follows.

`define c-constant` **SYMBOL** = ***expression;***

The following is an example of its usage.

```
define c-constant $max-size-of-people = 55;
```

5.7 Defining C Functions

C functions can be imported into Dylan using the following

`define c-function` ***dylan->c-name-specifier***
`(`***dylan->c-parameter-specifier ...***`) =>` ***c->dylan-type-specifier***

The following is an example of a definition of a C function.

```
define c-function xt-initialize => "XtInitialize"
  (program-name :: <byte-string> => <C/string>,
   app-name :: <byte-string> => <C/string>,
   resources :: <integer> => <C/int>,
   resource-count :: <integer> => <C/int>,
   argc :: <integer> <= <C/int>,
   argv :: <integer> => <C/int>) =>
  (widget :: <X/widget>);
```

5.8 Defining C Callbacks

Dylan functions can be exported to C using the following.

`export-as-c-function` ***c->dylan-name-specifier***
`(`***c->dylan-parameter-specifier ...***`) =>` ***dylan->c-type-specifier***

c->dylan-name-specifier:

STRING `=>` **SYMBOL**

c->dylan-parameter-specifier:

c->dylan-type-specifier
SYMBOL `::` ***c->dylan-type-specifier***

The following is an example of its usage.

```
export-as-c-function
    "z_x_dylan_callback_dispatcher" => x-dylan-callback-dispatcher
  (widget :: <X/widget>,
   client-data :: <C/int> => <integer>,
   call-data :: <X/callback-data>)
  reference: alien-x-dylan-callback-dispatcher ???;
```

5.9 Creating C Instances

this section needs to be revamped to reflect the new design

Alien instances are created using **make-alien** or **make-alien-array**.

make-alien *alien-type* ⇒ *object* [Function]

Creates an instance whose type is *alien-type*, and returns an object whose type corresponds to a pointer to that type. The pointer points to newly allocated memory.

make-alien-array *alien-type* #key *size* ⇒ *object* [Function]

Creates an array of size *size* that contains elements of type *alien-type*, and returns a pointer to enough space for those objects.

dylan-alien-address *alien-object* ⇒ *integer* [Function]

Returns an integer identifying the address in memory of the alien object. This is intended to be used for hashing aliens to preserve uniqueness, since this isn't done automatically in the system.

do-cast-as *pointer alien-type ⇒ object* [Function]

Casts the alien pointer *pointer* to the specified alien type.

Dylan's arithmetic and element functions will be overloaded to allow programmers to push pointers, and read and write their values. Here is an example of the use of pointer arithmetic from Xt:

```
define variable $x-arg-array = make-alien-array(<X/arg>, size: 50);

define method as-xt-args (initlist :: <sequence>)
  for (i from 0,
       args = initlist then tail(tail(args)),
       until: (empty? args),
        finally: values($x-arg-array, i))
    let arg = $x-arg-array + i;
    arg->name(arg) := first(args);
    arg->value(arg) := second(args);
  end;
end method;

define method xt-args (#rest initargs)
  as-xt-args(initargs)
end method;
```

As in LispWorks, the DylanWorks FFI libraries require that the FFI definitions be processed and then that the required C libraries be loaded. The function that does this is **ensure-alien-modules**.

ensure-alien-modules #rest *modules* [Function]

Ensures that the requested libraries are loaded. For example,

```
(ensure-alien-modules "Xm" "Xt" "X11")
```

6. Open Issues

Do we want to be able to subclass C++ classes in Dylan?

Do we want to be able to subclass Dylan classes in C++?

Do we want to allow Dylan generic functions to specialize on C++ classes?

Do we want to allow C++ generic functions to specialize on Dylan classes?

There is an issue around **export-as-alien-function**. SWM believes that, since one of our goals is to be able to produce Dylan libraries that can be linked with arbitrary C programs, that it is not really reasonable to expect that programmers will have to explicitly write **export-as-alien-function** for every entry point in a library. At the very least, it would be nice if there was some kind of semi-automated way of saying that everything in a library should be exported this way. Maybe even a modifier to **define method** or **define generic**.

Are the new prefix Dylan names for the alien import facilities OK? Strictly speaking, they don't define anything, but "define mumble" does seem nicer to us.

define alien-function needs a "language" argument, right? How about a "compiler" argument?

Is **ensure-alien-modules** really needed? Couldn't this be in the **define library** form or something? Or maybe we really want a **define alien-library** form?

7. Introduction to Creole

The purpose of this paper is to describe the functionality of DylanWorks Creole. Creole is a cross-language interface tool which will allow the user to interchange data with libraries that are written in other languages. This document describes the high-level application programmer interface (API) to Creole. Another document, Foreign Function Interface, will describe the machine level work required to implement Creole. Readers should also be familiar with Moon's paper, "MacDylan Creole"; it was the starting point for this white paper.

8. Design Ideas and Issues

- Use Apple's *Creole* proposal as a starting point.
- Evaluate new *Melange* proposal from CMU.
- Identify issues regarding interfacing with C++.
- Identify issues regarding calling Windows/NT libraries.
- Identify issues regarding architecture specific calling conventions.

9. Overall Functionality

The goal of DylanWorks Creole is to make it easy for users to build applications that are written in a combination of Dylan, C and C++. This may also include libraries that have a C or C++ interface that were written in any language. Creole performs five primary functions at the language level. These are interface importation, access paths, cross-language call, name mapping, and type mapping.

9.1 Interface Importation

Interface importation imports an interface defined in a C or C++ header file into a Dylan program. This enables a user program to call C or C++ functions from Dylan. This also allows the Dylan program to access global data items declared in the C or C++ header files. Apple proposes a syntax for the interface importation that is sufficient to import Apple's own C header files, but may not be quite sufficient for use with real C++ header files, or the less carefully maintained Unix (and possibly Microsoft) header files.

< Include import interface syntax here>

Here is a simple example of using Creole:

```
define interface
  #include "fred.h"
end interface;
```

Options are available to override the Creole's default behavior. The options allow a programmer to control the importation process in the following ways:

- Selectively imports part of the interface
- Explicitly control type mapping
- Explicitly control name mapping to avoid name conflicts caused by the differences between Dylan and C or C++. (case sensitivity and scoping rules)
- Control trade-offs between runtime memory consumption and dynamic functionality.

9.1.1 Open Issues with Importation

The following are issues and problems with the MacDylan Creole design:

- It is Mac-centric.
- It does not address calling C++ functions and accessing C++ data.
- It appears not to follow C standard rules for parsing #include files. The current document says, "If a **define interface** statement imports a header file a.h and a.h contains `#include "b.h"`, Creole does not import definitions in b.h." (p. 26, first paragraph).

  It may the case that this statement means that the included files should be parsed, but not imported. We need to find out what the intention of this statement is. At the very least, included files need to be recognized and parsed in order for the top-level header file to have the correct semantics.
- On most systems, there are predefined C or C++ macros like __STDC__ ; it is unclear whether the user would be required to define system predefined macros in order to get the include file to parse correctly.

  We recommend that we match the native compiler(s) CPP macros so the user does not need to worry about this issue.
- Default -I paths. CPP normally has a set of default include paths that are searched to find include files that are specified with the <> syntax. This ordered set normally includes `/usr/include` and can include multiple entries (especially for C++). For some compilers there are environment variables that add to or modified the include path search rules.

  We recommend that we add support for <> syntax and match the include path search rules for supported C and C++ compilers.
- Feature suggestion: minimal addition if the user specifies a specific function (or variable) from an include file bring in only the necessary parts of the include file. For example:

```
define interface
#include "string.h", // I would use <string.h> but this is not part
of the language yet
import: {"strncpy"};
end interface;
```

The prototype for `strncpy` is:

```
extern char * strncpy(char *s1, const char *s2,size_t n);
```

This would import the C function strcpy and the user defined type size_t, which is defined to to the type `int`, and discard the remaining contents of the file. This approach would also be useful for minimal or incremental recompilation systems. i.e. If the user added an additional prototype to a file it would not change the declaration of `strcpy,` so the application would not need to be recompiled.

- Dylan to C++ name mapping is much more difficult than Dylan to C name mapping. The C++ naming scheme encodes the types of the parameters into the function name. Most debuggers provide a set of functions (called something like `mangle` and `demangle`) that translate internal names to external names and vice-versa. C++ also has some very arcane rules for data type promotion/mapping when there is not an exact match of function signature. For example:

  `<insert examples here>`

  If we are going to support more than one C++ compiler each compiler may (probably will) have its own name mangling scheme. So, we will need to provide a separate mangle/demangle function for each compiler.

- A minor issue is that `define` only allows the definition of C macros to integers. However, it is possible in C to define a macro to a character string, for example,

  ```
  #define foo "this is a string"
  ```

  We have not seen this used in header files, but you never know what the user or library vendor may do.

  We suggest that this should be implemented if it is trivial, otherwise it should be documented as a limitation.

9.2 Access Paths

DylanWorks Creole will support the following access paths:

- External module
- Libraries
- Inlined machine code, if it is possible
- Others as needed by the target OS **<need help from NT person here to add to list>**

9.3 Cross-language Call

Cross-language call allows Dylan routines to call routines in another language and vice-versa. You can pass a pointer to a Dylan function as an argument to a C function. The C function can call back the Dylan function with an "alien-method" or a macro defined by the callback clause of **define interface**.

9.4 Name Mapping

Name mapping translates names of entities in another language into Dylan variable names in a specified module.

Creole allows you to specify the exact mapping of each name, select one of the several available sets of name translation rules using the **name-mapper:** option, or use the default set of name translation rules.

MacDylan provides the following name mapping options:

- Blah blah blah

9.4.1 Open Issues with Name Mapping

- A proposed enhancement is to allow the user to provide a function that would do the name mapping, such as for the following C prototype:

```
        prototype char * mapper(const char* dylan_name, char *buffer_p);
```

  where `dylan_name` is a pointer to a C-string and `buffer_p` is a pointer to an area of memory sufficient to return the mapped function name. If the function fails to perform the mapping or an error occurs while filling the buffer, a well-defined error condition should be defined.

9.5 Type Mapping

Type mapping translates C and C++ types into Dylan types.

Low-level facilities provide direct use of machine pointers and the raw bits pointed to by the machine pointers.

<This subject will be covered by the FFI paper>