MkImage Specification[†]
Version 0.2

Jonathan Shapiro, Ph.D.
The EROS Group, LLC.

February 26, 2007

Abstract

MkImage is the tool used to create installable application images, including the initial system bootstrap file image. It may be thought of as a link editor for *systems*, in much the way that `ld` is a link editor for object files. A `mkimage` input file contains a specification for what processes need to exist in the image, how they should be connected together by capabilities, and what (if any) external references need to be resolved in order for the image to be successfully incorporated by the installer.

This document defines the `mkimage` input specification language. It borrows heavily from the original `mkimage` specification from the EROS system.

Contents

[†] Copyright © 2007, Jonathan S. Shapiro.

1 Introduction

The `mkimage` utility creates an image for the Coyotos [1] installer. Through this tool, a developer can create a single image file that includes running processes, address spaces, and other objects. These various objects may be pre-initialized to refer to each other via capabilities. The input file can also contain unresolved capability references that are to be "filled in" at image load time.

More precisely, the "running" processes in a `mkimage`

output file are processes that are frozen at the instant before the execution of their first instruction. These are added to a "run list" that is started by the installer, or in the case of the initial system image, by the **IPL process**, whose job (in either case) is to actually get these processes into motion. It might therefore be more precise to say that these processes are *marked* to be run.

1.1 Theory of Operation

A `mkimage` script consists of a single module containing a sequence of statements. The first import of a module causes its statements to be executed in sequence, with several consequences:

1. Symbols defined within the module are assigned values.

2. A subset of these symbols (those designated as "exported") are made available for use in other modules.

3. Objects are allocated and initialized that become part of the system image file.

While `mkimage` can be used as a scripting language, it isn't designed for this purpose. A `mkimage` script is executed for the sake of its side effects on the image file.

For example, the `mkimage` script:

```
module example {
  export def a = 1 + 2;
  def b = make_process();
}
```

has three effects. The symbol `a` is bound to the value 3 and exported. A new, running process is allocated. The symbol `b` is bound this new process. Of these three effects, the important one is the allocation of the new process structure. At the end of the `mkimage` execution, the symbols will be gone, but this process structure and its contents will end up in the Coyotos installable image file. The reason to have a full scripting language is to provide flexibility in initializing the objects that will end up in the coyotos image.

A `mkimage` script can refer to capabilities that are not defined by any imported module. This mechanism is used to refer to capabilities that will later be supplied by the installer at image install time. When constructing an initial system image, there should be no unresolved references of this form.

1.2 Command Line

The `mkimage` command line takes the form:

```
mkimage
    [options]
    -t target
    specfile specfile
```

Where *target* is the target architecture. Other options understood by `mkimage` include:

-I *directory* Add *directory* to the end of the module resolution path.

-L *directory* Add *directory* to the end of the library search path.

-H *headerDir* Emit header files suitable for C and C++ programs into *headerDir* for each module named on the command line. The names of the header files are derived from the module names. A constant definition will appear in the header file for every member of every enumeration in the module that is qualified by the `header` keyword.

If this option is given, no binary Coyotos image file is emitted.

-o *output_file* Place the constructed system image in `output_file`. If this option is not given the default image file name is `mkimage.out`.

-v Process the input verbosely.

-V Report the `mkimage` version and exit.

–showparse Show the actions of the parser (for debugging).

–showlex Show the actions of the tokenizer (for debugging).

–showpasses Emit a list of `mkimage` compiler passes.

–stopafter *pass* Stop processing after *pass* has been run on the input (for debugging).

–dumpafter *pass* Show the state of the AST after *pass* as an XML dump (for debugging).

The `mkimage` utility is not designed for incremental use. Like `ld`, its job is to create the final image for a system. There is (at the moment) no conceptual equivalent to creating a relocatable, reusable library.

1.3 Specification Conventions

The primary conventions used in this document concern the presentation of grammar rules. In grammar rule specifications, non-terminals are presented in lowercase italics,

categorical terminals are shown in normal face, and literal terminals appear in bold face or within single-quote characters. White space is permitted between tokens. `mkimage` is case sensitive; all `mkimage` keywords are lowercase.

Except when quoted, the characters { and } indicate metasyntatic grouping following the customary representation of EBNF grammars. The superscript characters *****, +, and **?** indicate, respectively, zero or more, one or more, or an optional item or group. In Except when quoted, the character ":" indicates the separation between a non-terminal and its EBNF definition. For categorical terminals, sets of input characters are abbreviated by enclosing them within square brackets. Within such a set, the character '-' denotes an inclusive lexical sequence according to the usual conventions of regular expressions.

2 Lexical Matters

The `mkimage` input character set is the UTF-8 encoded Unicode character set, as defined by the Unicode 4.1.0 standard [2].

Input units of compilation are defined to be encoded using the Unicode character set as defined in version 4.1.0 of the Unicode standard [2], using Normalization C. All keywords and syntactically significant punctuation fall within the ISO-LATIN-1 subset, and the language provides for ISO-LATIN-1 encodable "escapes" that can be used to express the full Unicode character code space in character and string literals.

Tokens are terminated by white space if not otherwise terminated. For purposes of input processing, the characters *space* (U+0020), *tab* (U+0009), *carriage return* (U+000D), and *linefeed* (U+000A) are considered to be white space.

Input lines are terminated by a linefeed character (U+000A), a carriage return (U+000D) or by the two character sequence consisting of a carriage return followed by a line feed. This is primarily significant for comment processing and diagnostic purposes, as the rest of the language treats linefeeds as white space without further significance.

2.1 Comments

`mkimage` supports two comment formats derived from C and C++. Any sequence beginning with `//` and ending with the next newline is a comment. Any character sequence beginning with `/*` and ending with the next `*/` is a comment. No beginning of comment appearing within a comment is considered lexically significant. For purposes of input tokenization, a comment is considered to be whitespace, and terminates any input token. Comments therefore cannot be used for "token splicing."

2.2 Reserved Words

The following identifiers are syntactic keywords or literal constants, and may not be rebound:

```
capreg      def         do
else        enum        export
external    false       for
header      if          import
lambda      module      new
print       return      true
while       using
```

Reserved words shown in italics are not currently implemented, but are reserved for future use.

2.3 Identifiers

A `mkimage` identifier may start with any "identifier character" (UNICODE 4.1.0 character class XID_Start), followed by any number of optional "identifier continue characters" (UNICODE 4.1.0 character class XID_Continue). For this purpose, the underscore character ('_', U+005F) is considered to be an identifier character, and the hyphen character ('-', U+) is considered to be an identifier continue character. Any such sequence that is not a reserved word is an identifier. Identifiers beginning with two leading underscores are reserved.

A hyphen, if present, may not be the leading character of an identifier. Hyphens are converted to underscores when data constant bindings are exported for use in C or C++ header files.

For the sake of compatibility with existing file systems, identifiers appearing in `mkimage` module names are restricted to characters that fall within the ISO-LATIN-1 subset of the general identifier specification.

The following regular expression describes those identifiers using only characters from the ISO-LATIN-1 subset. This subset is commonly used because such identifiers can be directly exported to languages such as C.

```
Ident: [a-zA-Z_][-a-zA-Z0-9_]*
```

2.4 Literals

2.4.1 Integer Literals

The general form of an integer literal is an octal, decimal, or hexidecimal constant:

```
IntLit: [1-9][0-9]*
IntLit: 0[0-7]*
IntLit: 0x[0-9a-zA-Z]+
```

Integer literals beginning with the digit zero (0) are interpreted in base 8 (octal). Integer literals starting with "0x" are interpreted in base 16 (hexadecimal). Hexidecmal digits are interpreted with the customary hexadecimal valuations. The letters may appear in either lowercase or uppercase. It is an error for a digit to be present whose value as a digit is greater than or equal to the specified base.

2.4.2 String Literals

String literals (`String`) are written within double quotes, and may contain printable characters *excluding* backslash ("\"). They may also contain spaces (U+0020). The set of printable characters consists of any character specified in the UNICODE 4.1.0 standard *except* those with general categories "Cc" (control codes) "Cf" (format controls), "Cs" (surrogates), "Cn" (unassigned), or "Z" (separators). That is, any printable character, excluding spaces.

```
StringLit: /* as described above
```

Within a string, the backslash character ("\") is interpreted as beginning an encoding of a specially embedded character. The character following the "\" is either a single-character embedding or a curly brace character "{" identifying the start of a UNICODE character embedding. The legal forms and their meanings are:

\n	Linefeed
\r	Carriage Return
\t	Horizontal Tab
\b	Backspace
\s	Space
\f	Formfeed
\"	Double quote
****	Backslash
\{U+*digits*}	Unicode code point, hexidecimal *digits*.

I Core Language Definition

3 Structure of a Compilation Unit

A `mkimage` unit of compilation consists of a single module definition, which in turn consist of a sequence of module-level statements and definitions:

```
UOC: module modname '{' modstmt+ '}'
modname: Ident {'.' Ident}*
modstmt:
   | export def_stmt
   | export Ident {',' Ident}* ';'
   | import Ident '=' modname ';'
   | stmt
```

In contrast to "inner" definitions, definitions introduced at module scope may be exported, making them importable from other modules.

3.1 Import and Export

Export The `export` keyword indicates that a symbol bound at top level in a module should be visible to importing modules. When interpretation of the module is completed, and the top-level environment is fully defined, a selective copy of the elements of this environment is made to create a "public" environment containing only the exported symbols. This is the environment that is obtained by a subsequent `import` statement.

Import The **import** statement has two effects. If the imported module has not yet been loaded and intepreted, this is done at the time of the first import. A given module will be imported at most once during an execution of `mkimage`.

The left-hand identifier specified in an `import` gives a local identifier to which the public environment of the imported module is bound. Thus, if a is defined to 1 and exported by module `coyotos.SpaceBank`, then the statements:

```
import sb = coyotos.SpaceBank;
print sb.a;
```

displays the value "1".

3.2 Defining Statements

The defining statments introduce variables, procedures, and enumerations:

```
def_stmt: def Ident '=' expr ';'
  | def Ident ({Ident {',' Ident}*}?')'
      block ';'
  | header? enum '{' enumDef+ '}'
  | header? enum Ident '{' enumDef+ '}'
  | header?capreg '{' enumDef+ '}'
  | header?capreg Ident '{' enumDef+ '}'
enumDef : Ident {'=' expr}*
```

The first **def** form associates an initial the value with an identifier and binds that identifier in the current scope.

The second **def** form introduces a procedure. When called, the body of the procedure will be executed in an environment consisting of its lexically preceding definitions within the module and the formal parameters of the procedure. Inner procedures are legal, and can be returned as values.

The **enum** form introduces enumerated constant values. The values of these identifiers cannot be modified. Enumerated constants are supported in the language because there are certain values that mkimage and it's constructed programs must agree on. In particular, enumerations are used to define slot numbers in the "tools" capability page of each process.

The **capreg** form introduces enumerated constant values that are to be used as capability register numbers. For purposes of use inside a mkimage file, there is no difference between a **capreg** enumeration and an **enum**.

When an **enum** or **capreg** definition is market by the **header** keyword, definitions for the enumerated constants will be emitted to the corresponding C header file when header files are generated. Constants defined in an **enum** enumeration will be emitted as integer literals, while constants defined in a **capreg** enumeration will be emitted as constant structure declarators of type capreg_t.

When an expression appears on the right hand side of an **enumDef**, it is subject to several constraints:

- The only functions that can be called are arithmetic functions and unary negation.

  The only identifiers that can be referenced are identifiers defined by earlier **enumDef** forms.

The combined effect of these constraints is that enumeration definitions may make reference to, and compute values based on, other enumeration definitions, but they do not require general interpretation to occur. This is necessary to support header file generation. At the time header files are generated the corresponding binary programs do not exist, so attempts to execute (e.g.) **loadimage**() would fail.

The value of a defining statement is *unit*.

3.3 Control Flow Statements

The statement:

```
if_stmt: if '(' expr ')' block { else block
  | if '(' expr ')' block { else if_stmt
stmt: if_stmt
```

provides conditional execution with the customary meaning. The value of this statement is the value of the executed block, or *unit* if the test expression produces *false* and no else clause is given. The expression in the test position should generate either an integer or a boolean result. If the result is an integer, the value zero is interpreted as a false condition result and non-zero is interpreted as a true condition result.

The statement:

```
return_stmt: return expr ';'
```

causes a value to be returned from the nearest enclosing procedure. This statement is not legal at module scope. Any statement following a return statement will not be executed.

The statement:

```
while_stmt: while '(' expr ')' block
```

Causes the sequence of statements within the block to be executed as long as the value of *expr* evaluates to true (see if statement, above). The return value of this statement is the value of the last iteration of executed block, or Unit if the loop is not executed at all.

The statement:

```
do_stmt: do block while '(' expr ')' ';'
```

Causes the sequence of statements within the block until the value of *expr* evaluates to true (see if statement, above). The return value of this statement is the value of the last iteration of executed block.

3.4 General Statements

The traditional syntax for assignment is accepted, and statements may be executed for side effects:

```
stmt: expr '=' expr ';'
stmt: expr ';'
```

The left hand side of an assignment must be an "lvalue". That is: an identifier bound in the current environment, a field of an object, or a slot of an array.

Sequences of statements may be surrounded by braces to form a statement block:

```
block: '{' block_stmt+ '}'
block_stmt: stmt
  | return_stmt
  | def_stmt
```

A block is not a legal statement. The value of a block is the value of the last executed statement within the block.

Diagnostics may be obtained during execution through two `print` statements:

```
stmt: print expr
  | print '*' expr
```

The first prints the value of the expression *expr*. If *expr* is a capability-valued expression denoting an object, the second prints the fields of the object in human-readable form.

The print statements *may* be dropped shortly in favor of comparable built-in procedures.

4 Expressions

`mkimage` provides both the conventional "basic" expressions and a number of expressions that are particular to the problem of system image construction.

4.1 Basic Expressions

`mkimage` supports the traditional infix arithmetic, comparison, and boolean negation operators with the usual precedence:

```
expr: '-' expr
expr: expr arithop expr
expr: expr cmpop expr
expr: '!' expr
arithop: '-' | '+' | '/' | '*' | '%'
cmpop: '<' | '>' | '<=' | '>='
  | '==' | '!='
```

Arbitrary-precision integer literals, double-precision floating point values, and strings are valid expressions:

```
expr: Int | Float | String
```

The identifiers `true` and `false` are built-in literals:[1]

```
expr: true | false
```

---
[1] True and false are not currently implemented.

Procedure calls are expressions written in the usual way:

```
expr: expr '(' expr* ')'
```

4.2 Object Allocation Expressions

Processes, GPTs, Endpoints, Pages, CapPages, and Banks are primitive objects. Creating one entails reserving real storage in the output image file. In consequence, allocation of these is supported by syntax:

```
expr: new Bank '(' expr ')'
expr: new CapPage '(' expr ')'
expr: new GPT '(' expr ')'
expr: new Endpoint '(' expr ')'
expr: new Page '(' expr ')'
expr: new Process '(' expr ')'
```

The *expr* argument must be a bank capability.

4.3 Vector and Environment Expressions

If `env` is a environment-valued expression, and `ident` is a field of the corresponding object, then: `env.ident` is an lvalue expression denoting the location of that field. When this appears in a use-occurrence, it is automatically promoted to the *value* of the field.

```
expr: expr '.' Ident
```

For each of the objects defined in the Coyotos specification, if *expr* is a capability-valued expression naming one of these objects then *expr* is also an environment-valued expression whose identifiers are the fields of the object. For example:

```
;references the fault code slot:
processCap.faultCode
;references the GPT capability array:
gpt.cap
```

If `evec` is an expression denoting an array, and `ndx` is an integer-valued expression, then: `evec[ndx]` is an lvalue expression denoting a slot in that array (or raises an exception if bounds are violated). When this appears in a use-occurrence, it is automatically promoted to the *value* of the field.

```
expr: expr '[' expr ']'
```

For example:

```
    ;references to a capability register:
    processCap.capReg[ndx]
    ;references to a GPT capability slot:
    gpt.cap[2]
```

Environment-valued and vector-valued expressions are obtained by calling various built-in library procedures within the `mkimage` runtime. A capability page capability is a vector-valued expression. The `cap` field of a GPT is a vector-valued expression.

4.4 Block Expressions

A statement block is an expression whose value is the value of the last statement. Note that this subsumes the need for a local binding construct:

```
    def a = { def i = 1; i + 3; };
```

II   Built-In Procedures

5   Kernel Object Constructors

For each kernel capability defined by the Coyotos kernel, there is a corresponding constructor function in the runtime library that returns this capability. In some cases the capability is a "miscellaneous" capability that does not designate an object. In other cases the capability designates an object and a call to the constructor function has the side effect of adding an object of that type to the coyotos image that is under construction:

Window(*offset*) Returns a window capability into the background space with the specified integer-valued offset.

LocalWindow(*offset*, *slot*) Returns a local window capability into the space named by *slot* with the specified integer-valued offset.

enter(*cap*, *pp*) Fabricates an entry capability to the endpoint named by *cap*. The protected payload value of the fabricated capability will be *pp*.

NullCap() Returns a null capability.

KeyBits() Returns the KeyBits capability.

Discrim() Returns the Discrim capability.

Range() Returns the Range capability.

Sleep() Returns the Sleep capability.

Range() Returns the Sleep capability.

IrqCtl() Returns the IrqCtl capability.

SchedCtl() Returns the SchedCtl capability.

Checkpoint() Returns the Checkpoint capability.

ObStore() Returns the ObStore capability.

IoPerm() Returns the IoPerm capability.

PinCtl() Returns the PinCtl capability.

There is also a procedure for duplicating an existing object:

dup(*bank*, *cap*) Given a capability *bank* to a space bank, duplicates the object named be *cap* and returns a capability to the duplicate.

6   Capability Manipulation

The following built-in procedures manipulate and/or modify existing capabilities:

readonly(*cap*) Returns a read-only variant of the passed capability *cap* (sets the RO restriction bit).

weaken(*cap*) Returns a read-only and weak variant of the passed capability *cap* (sets the WK, RO restriction bits).

noexec(*cap*) Returns a non-executable variant of the passed capability *cap* (sets the NX restriction bit).

opaque(*gpt-cap*) Given a GPT capability *gpt-cap*, returns an opaque capability to the same GPT.

guard(*cap*, *guard* [, *l2g*]) Given a memory object capability *cap*, returns a capability with the same permissions whose guard value is set to (*guard*$>>$*l2v*). If *l2g* is not specified, the capability's existing *l2g* value is used.

7   Space and Object Manipulation

There are a variety of functions for manipulating address spaces. It is probably advisable to experiment with these using *print_tree* and *print_space* to fully understand what they do.

readfile(*bank*, *string*) Given a capability *bank* to a space bank, and a file name *string*, reads the file whose path is provided in *string* and returns an appropriate address space capability. This routine will allocate any GPT and page structures necessary to fabricate the address space.

loadimage(*cap*, *string*) Given a capability *bank* to a space bank, loads an executable file from the file named by *string* into an address space in executable form. The file name should name an ELF executable. The result is an address space that is populated exactly as needed to contain the loadable sections specified in the ELF file's program header. This routine will allocate any GPT and page structures necessary to fabricate the address space.

In contrast to *readfile*, this procedure returns an environment-valued result. The environment will have bound identifiers;

pc An integer-valued expression containing the binary's entry point.

space A capability-valued expression containing the fabricated address space.

guarded_space(*bank*, *cap*, *guard* [, *l2g*]) Given a capability *bank* to a space bank, and a capability *cap* to an existing address space, returns a new capability to the same space having the specified *guard* and *l2g* values. If *l2g* is not provided, the current *l2g* value of the address space capability is used.

In contrast to *guard*, which acts on the capability alone, *guarded_space* may insert additional GPT structures in order to satisfy a guard request that would not fit trivially into the guard field.

insert_subspace(*bank*, *spc*, *subspc*, *offset* [, *l2arg*]) Given capabilities a capability *bank* to a space bank, and capabilities *spc* and *subspc* that name existing address spaces, inserts the address space named by *subspc* into the address space named by *spc* at the specified *offset*. If the effective size of *subspc* exceeds the effective size of *spc*, returns *subspc*.

The *l2arg* value, if supplied, gives the *effective size* of the subspace being inserted, which may be either larger or smaller than the actual size of the subspace. In this case, the subspace is inserted into the containing space *as if* it were of the size indicated by *l2arg*.

print_tree(*cap*) Given an address space capability *cap*, recursively prints the objects comprising the address space in raw form.

print_space(*cap*) Given an address space capability *cap*, recursively prints the objects comprising the address space. The output is designed to provide a picture of what pages are mapped where and with what permissions.

fillpage(*cap*, *uint8*) Fills the page referenced by *cap* with the 8-bit value *uint8*.

set_page_uint64(*cap*, *addr*, *uint64*) Writes the 64-bit value *uint64* at offset *addr* in the page referenced by *cap*. *addr* must be a multiple of 8 less than the target's page size. The value is written in the target's endianness.

A   Change History

This section is an attempt to track the changes to this document by hand. It may not always be accurate!

This section is non-normative.

A.1   Version 0.2

Complete rewrite reflecting new implementation.

A.2   Version 0.1

Initial capture.

References

[1] J. S. Shapiro, Jonathan W. Adams, Eric Northup, M. Scott Doerrie, Swaroop Sridhar, Neal H. Walfield, and Marcus Brinkmann. *Coyotos Microkernel Specification*, 2007, available online at www.coyotos.org.

[2] Unicode Consortium. The Unicode Standard, version 4.1.0, defined by *The Unicode Standard Version 4.0*, Addison Wesley, 2003, ISBN 0-321-18578-1, as amended by *Unicode 4.0.1* and by *Unicode 4.1.0*. http://www.unicode.org.