Kconfig macro language

Concept

The basic idea was inspired by Make. When we look at Make, we notice sort of two languages in one. One language describes dependency graphs consisting of targets and prerequisites. The other is a macro language for performing textual substitution.

There is clear distinction between the two language stages. For example, you can write a makefile like follows:

```
APP := foo
SRC := foo.c
CC := gcc

$(APP): $(SRC)
        $(CC) -o $(APP) $(SRC)
```

The macro language replaces the variable references with their expanded form, and handles as if the source file were input like follows:

```
foo: foo.c
        gcc -o foo foo.c
```

Then, Make analyzes the dependency graph and determines the targets to be updated.

The idea is quite similar in Kconfig - it is possible to describe a Kconfig file like this:

```
CC := gcc

config CC_HAS_FOO
        def_bool $(shell, $(srctree)/scripts/gcc-check-foo.sh $(CC))
```

The macro language in Kconfig processes the source file into the following intermediate:

```
config CC_HAS_FOO
        def_bool y
```

Then, Kconfig moves onto the evaluation stage to resolve inter-symbol dependency as explained in kconfig-language.rst.

Variables

Like in Make, a variable in Kconfig works as a macro variable. A macro variable is expanded "in place" to yield a text string that may then be expanded further. To get the value of a variable, enclose the variable name in $( ). The parentheses are required even for single-letter variable names; $X is a syntax error. The curly brace form as in ${CC} is not supported either.

There are two types of variables: simply expanded variables and recursively expanded variables.

A simply expanded variable is defined using the := assignment operator. Its righthand side is expanded immediately upon reading the line from the Kconfig file.

A recursively expanded variable is defined using the = assignment operator. Its righthand side is simply stored as the value of the variable without expanding it in any way. Instead, the expansion is performed when the variable is used.

There is another type of assignment operator; += is used to append text to a variable. The righthand side of += is expanded immediately if the lefthand side was originally defined as a simple variable. Otherwise, its evaluation is deferred.

The variable reference can take parameters, in the following form:

```
$(name,arg1,arg2,arg3)
```

You can consider the parameterized reference as a function. (more precisely, "user-defined function" in contrast to "built-in function" listed below).

Useful functions must be expanded when they are used since the same function is expanded differently if different parameters are passed. Hence, a user-defined function is defined using the = assignment operator. The parameters are referenced within the body definition with $(1), $(2), etc.

In fact, recursively expanded variables and user-defined functions are the same internally. (In other words, "variable" is "function with zero argument".) When we say "variable" in a broad sense, it includes "user-defined function".

Built-in functions

Like Make, Kconfig provides several built-in functions. Every function takes a particular number of arguments.

In Make, every built-in function takes at least one argument. Kconfig allows zero argument for built-in functions, such as $(filename), $(lineno). You could consider those as "built-in variable", but it is just a matter of how we call it after all. Let's say "built-in function"

here to refer to natively supported functionality.

Kconfig currently supports the following built-in functions.

- $(shell,command)

  The "shell" function accepts a single argument that is expanded and passed to a subshell for execution. The standard output of the command is then read and returned as the value of the function. Every newline in the output is replaced with a space. Any trailing newlines are deleted. The standard error is not returned, nor is any program exit status.

- $(info,text)

  The "info" function takes a single argument and prints it to stdout. It evaluates to an empty string.

- $(warning-if,condition,text)

  The "warning-if" function takes two arguments. If the condition part is "y", the text part is sent to stderr. The text is prefixed with the name of the current Kconfig file and the current line number.

- $(error-if,condition,text)

  The "error-if" function is similar to "warning-if", but it terminates the parsing immediately if the condition part is "y".

- $(filename)

  The 'filename' takes no argument, and $(filename) is expanded to the file name being parsed.

- $(lineno)

  The 'lineno' takes no argument, and $(lineno) is expanded to the line number being parsed.

Make vs Kconfig

Kconfig adopts Make-like macro language, but the function call syntax is slightly different.

A function call in Make looks like this:

```
$(func-name arg1,arg2,arg3)
```

The function name and the first argument are separated by at least one whitespace. Then, leading whitespaces are trimmed from the first argument, while whitespaces in the other arguments are kept. You need to use a kind of trick to start the first parameter with spaces. For example, if you want to make "info" function print " hello", you can write like follows:

```
empty :=
space := $(empty) $(empty)
$(info $(space)$(space)hello)
```

Kconfig uses only commas for delimiters, and keeps all whitespaces in the function call. Some people prefer putting a space after each comma delimiter:

```
$(func-name, arg1, arg2, arg3)
```

In this case, "func-name" will receive " arg1", " arg2", " arg3". The presence of leading spaces may matter depending on the function. The same applies to Make - for example, $(subst .c, .o, $(sources)) is a typical mistake; it replaces ".c" with " .o".

In Make, a user-defined function is referenced by using a built-in function, 'call', like this:

```
$(call my-func,arg1,arg2,arg3)
```

Kconfig invokes user-defined functions and built-in functions in the same way. The omission of 'call' makes the syntax shorter.

In Make, some functions treat commas verbatim instead of argument separators. For example, $(shell echo hello, world) runs the command "echo hello, world". Likewise, $(info hello, world) prints "hello, world" to stdout. You could say this is _useful_ inconsistency.

In Kconfig, for simpler implementation and grammatical consistency, commas that appear in the $( ) context are always delimiters. It means:

```
$(shell, echo hello, world)
```

is an error because it is passing two parameters where the 'shell' function accepts only one. To pass commas in arguments, you can use the following trick:

```
comma := ,
```

```
$(shell, echo hello$(comma) world)
```

Caveats

A variable (or function) cannot be expanded across tokens. So, you cannot use a variable as a shorthand for an expression that consists of multiple tokens. The following works:

```
RANGE_MIN := 1
RANGE_MAX := 3

config FOO
        int "foo"
        range $(RANGE_MIN) $(RANGE_MAX)
```

But, the following does not work:

```
RANGES := 1 3

config FOO
        int "foo"
        range $(RANGES)
```

A variable cannot be expanded to any keyword in Kconfig. The following does not work:

```
MY_TYPE := tristate

config FOO
        $(MY_TYPE) "foo"
        default y
```

Obviously from the design, $(shell command) is expanded in the textual substitution phase. You cannot pass symbols to the 'shell' function.

The following does not work as expected:

```
config ENDIAN_FLAG
        string
        default "-mbig-endian" if CPU_BIG_ENDIAN
        default "-mlittle-endian" if CPU_LITTLE_ENDIAN

config CC_HAS_ENDIAN_FLAG
        def_bool $(shell $(srctree)/scripts/gcc-check-flag ENDIAN_FLAG)
```

Instead, you can do like follows so that any function call is statically expanded:

```
config CC_HAS_ENDIAN_FLAG
        bool
        default $(shell $(srctree)/scripts/gcc-check-flag -mbig-endian) if CPU_BIG_ENDIAN
        default $(shell $(srctree)/scripts/gcc-check-flag -mlittle-endian) if CPU_LITTLE_ENDIAN
```