A User's Guide to extract

R. Clint Whaley *

July 31, 2015

Abstract

Extract is a software management tool which allows the user to store different versions of the same code or several related codes in one file. Interleaving of routines allows the user to have code such that when one version is updated, all versions are changed. Many capabilities of C's preprocessor are emulated, but the code produced by extract is much more readable. Extract also has some scripting abilities not present in cpp, such as looping structures and integer arithmetic. The code developer keeps his code in basefiles (see below) but gives the extracted files to users. A successful use of extract will result in files that no one can tell came from extract.

*Dept. of Computer Science, University of Texas at San Antonio, San Antonio, TX, 78249, whaley@cs.utsa.edu

Contents

1 Basics

Typing `extract -help` prints out usage information. In its most basic form, extract is used to interleave two or more related routines. Extract takes input from a file refered to as the *basefile* and creates a compilable source code called the *newfile* or *outfile*. Code that is included in the outfile is chosen based on user selectable *keys*. Extract allows the user an arbitrary number of these keys. In the basefile, lines containing keys are indicated by *@<keyhandle>*, so a basefile *keyline* is of form:

`@<keyhandle> [!]  match1 match2 ...matchN`

Keyhandle is the name of the key you are matching on, which could be, for instance, the target architecture, or the type of data used, etc. The `!` is a **not** operator: it reverses the normal meaning of keys – in other words, include the following lines if there is NOT a match. An example will go a long way to clearing up how this works. Below is shown a function for figuring the sum of a vector. There is one key used, which controls the data type. Assume this is in a file called `sum.base`.

```
@TYPE double
       DOUBLE PRECISION FUNCTION SUM(N, X)
       DOUBLE PRECISION X(*), TOT
@TYPE integer
       INTEGER FUNCTION SUM(N, X)
       INTEGER X(*), TOT
@TYPE !
       INTEGER N, I

       DO I = 1, N
          TOT = TOT + X(I)
10     CONTINUE

       SUM = TOT

       RETURN
       END
```

If we typed `extract -b sum.base -o isum.f type=integer`, we would get a file called `isum.f` which would contain:

```
       INTEGER FUNCTION SUM(N, X)
       INTEGER X(*), TOT
       INTEGER N, I

       DO I = 1, N
          TOT = TOT + X(I)
10     CONTINUE

       SUM = TOT
```

```
RETURN
END
```

The way this works is that extract scans through the basefile until it finds an @keyhandle that matches one of those given on the command line (in our example, the keyhandle was `TYPE`). When it finds such a keyline, it checks to see if the key given on the command line (call it the *commandkey* for convenience) matches a key on the keyline. If so, it continues to dump the output to the outfile. If the commandkey is not found on the keyline, it skips all lines until it finds a keyline with the correct keyhandle that has matching keys.

So in the above example, extract read the 1st line and found the keyhandle `@TYPE`, so it checked to see if `integer` (our commandkey) was on the keyline (in this case, `@TYPE double`). Since it was not, the following 2 lines were skipped, until it reached the line `@TYPE integer`. At that point, it had a keymatch, so the following two lines were included. Finally, it reached the line `@TYPE !`. This line had no keys on it. Therefore, no commandkey could be matched. However, it had the not operator applied to it. Not no match = match, so every file created from this basefile would get all the rest of the lines.

2 Command line options explained

The commandkeys on the command line have a limited wildcard facility, using the omnipresent @. For example `sys=@het@` will have keymatch for any key with `het` imbedded in it. The wildcard characters @ may only be used at beginning or end of the keys, i.e. `sys=@obbi type=int@ OS=@nix@` are all legal usages, but `sys=bo@b` is not.

Flags, mneumonics (or keyhandles) and keys are not case sensitive. The basefile and newfile parameters are case sensitive. If no input/output file is supplied, stdin/stdout is assumed. For all flags (excepting the commandline macro definition), adding the optional `0` to the end of the flag nullifies any previous setting. Default settings for all commandline options may be setup in the file `~/.extractrc`. The syntax of this file is just that of a commandline extract call, without the executable name (e.g. `-o bob.out -b bob.base -caseU -langC`). Flags are read in from left to right, so `-caseU -case0` results in no change of case.

The possible extract flags are:

- `-b <basefile>` The input basefile comes from the file <basefile>, instead of standard in. The user can specify `-b stdin`, in order to explicitly invoke standard in.
- `-o <outfile>` The output goes to file <outfile>, instead of standard out. The user can specify `-o stdout`, in order to explicitly invoke standard out.
- `-case[0,U,L]`
  - `-case0` Case is unchanged.
  - `-caseU` Everything but quoted strings upcased.
  - `-caseL` Everything but quoted strings lowcased.
- `-RepTab[0,#]`

- `-RepTab0` Do not replace tabs with spaces.
- `-RepTab` Replace tabs with 8 spaces.
- `-RepTab#` Replace tabs with # spaces.

- `-Remtblank[0]` Remove trailing blanks. Trailing blanks are all white-space characters after the last non-whitespace, and before the carriage return of a line.

- `-LAPACK[0,1,2]` Output code in butt-ugly LAPACK style:
  - `-LAPACK0` Take no LAPACK formatting action.
  - `-LAPACK1` Enforce the following LAPACK styles :
    1. All comments have * in first column.
    2. All but comments and strings are upcased.
    3. All blank lines become blank comment lines.
    4. Use only $ for continuation character.
    5. Labels are right justified to column 5
  - `-LAPACK2` Same as level 1, but also changes comment lines of

    ```
    *  -------
    *  comment
    *  -------
    ```

    to

    ```
    *
    *  comment
    *
    ```
  - `-LAPACK3` Use F90 syntax instead of f77. So:
    1. All comments have ! in first column.
    2. All blank lines become blank comment lines.
    3. Use only & for continuation character.
    4. Labels are right justified to column 5

- `-verb[0,1,2,3]` Vary extract's verbosity:
  - `-verb0` Extract prints nothing except warning and error messages.
  - `-verb1` Print begin and end extract messages from extract called from command line.
  - `-verb2` Print begin and end extract messages from basefile extracts as well.
  - `-verb3` Extract prints every I/O action: the opening and closing of all files, etc.

- `-lnlen[0,#]` Set the length for extract to break lines or warn about too long a line to #.

- `-llwarn[0,1,2]` Controls whether the extract issue line length warnings:

- `-llwarn0` No warnings are issued
- `-llwarn1` If the output language is Fortran77, comment lines do not generate warnings. For other languages, same as level 2 (this is because a comment line of length greater than 71 is not an error in Fortran77)
- `-llwarn2` Any line exceeding LnLen generates the error.

If LnLen is not set (or is set to 0) and this flag is set to 1 or 2, LnLen is assumed to be 71 for F77, and 80 for all other output languages.

- `-fmode[0,Q, A]` Output file mode:
  - `-fmode0` Overwrite file if it exists.
  - `-fmodeA` append to file if it exists.
  - `-fmodeQ` Query user before overwriting existing file.
- `-lang[0,F,C]` What language should be assumed: fortran 77 (-lang0 or -langF), or C (-langC).
- `-def <handle> "<replacement>"` Define a BLACS macro that will exist for the life of this extract only.
- `-addkeys[0]` When this flag is set, keys are always inherited, even when new ones are defined.
- `-punymac[0]` When this flag is set, the macros defined in the file(s) extracted by a `@extract` command are popped off before returning to the extracting file.
- `-trans "Ln#1, ..., Ln#N"` This option translates extracted file line numbers into basefile line numbers. I have temporarily disabled it.
- `-indent <col> <nspaces>` commandline way of doing indentation as described in Section 3.8.
- `no@[0,<basefile command>]` Allows the user to turn off basefile commands, so that extract no longer recognizes them. `no@all` turns of all commands.

Some examples should help. If you wanted to extract from `bob.base`, and upcase all output and remove trailing blanks, with the outfile being printed to standard out, you would type:

```
extract -caseU -remtblank -b bob.base OS=unix type=integer
```

Translating line number could be accomplished by a call of form:

```
extract -trans "5 2 44 63" -b blacs.base -o igsum2d.f type=real OS=VMS mach=VAX
```

.

2.1 Return value of extract

Extract returns a value of 1 if the following errors occur:

- Untranslatable key input on command line.
- Can't open required file.
- Can't close file.
- Not enough system memory.
- Command line input incorrect.
- @abort is called.

This is so that makefiles and shell scripts can tell if an error has occured. Otherwise, extract returns a value of 0.

3 Basefile commands

Extract has other commands that can be given inside of basefiles to ease the programmers burden. All these commands are indicated by beginning with an @ which is the first non-whitespace character on a line. The @ sign was chosen as the beginning character because it is illegal for most programming languages, and thus will cause an error if inadvertantly left in extracted code. The character ◇ is used in the text to indicate a significant whitespace that may not be omitted. Anywhere it is shown, there must be at least one whitespace, and there may be more if the user wishes.

The basefile commands are not case sensitive, although their arguments may be. For example, `@define type @double@` and `@DEFINE type @double@` are equivalent, but `@define TYPE @DOUBLE@` is not.

3.1 Macro usage

Extract has the ability to define macros, much like cpp. Macros are defined on a stack, and thus multiple definitions of a single handle are legal.

3.1.1 `@define`

Extract has a crude macro facility. A macro is defined by
`@define◇<handle>◇@<replacement string>@`. A macro is invoked by using its handle in the text, in the following way: `@(handle)`. Both the handle and replacement string are case sensitive. Macros defined by the user may not contain whitespaces, @ signs, or parenthesis. Here is our simple example we have seen before, rewritten to use macros:

```
@TYPE SREAL
   @define pre @S@
   @define type @REAL@
@TYPE DREAL
```

```
   @define pre @D@
   @define type @DOUBLE PRECISION@
@TYPE SCPLX
   @define pre @C@
   @define type @COMPLEX@
@TYPE DCPLX
   @define pre @Z@
   @define type @DOUBLE COMPLEX@
@TYPE !
       @(type) FUNCTION @(pre)SUM(N, X)
       @(type) X(*), TOT
       INTEGER N, I

       DO I = 1, N
          TOT = TOT + X(I)
10     CONTINUE

       @(pre)SUM = TOT

       RETURN
       END
```

Assuming this is in the file `tst.base`, and we issued the command `extract -b tst.base type=dreal`, extract would output the following to standard out:

```
       DOUBLE PRECISION FUNCTION DSUM(N, X)
       DOUBLE PRECISION X(*), TOT
       INTEGER N, I

       DO I = 1, N
          TOT = TOT + X(I)
10     CONTINUE

       DSUM = TOT

       RETURN
       END
```

3.1.2 `@undef`

Extract macros are actually based on a stack, so that if something is defined N times, the N'th definition is used, until `@undef` is called, at which point the N-1 definition is used. The syntax is `@undef◇<handle>`. Example:

```
   @define person @Victor Eijkhout@
   @define person @Clint Whaley@
```

```
Person 1: @(person)
   @undef person
Person 2: @(person)
   @undef person
No Macro: @(person)
```

Extracted code:

```
Person 1: Clint Whaley
Person 2: Victor Eijkhout
No Macro: @(person)
```

3.1.3 `@undefall`

If a handle is provided, this command undefines all instances of that macro (the syntax for this usage is thus `@undefall◇<handle>`). If no handle is provided, it undefines all macros defined in this basefile, and other basefiles called from this one (the syntax for this is simply `@undefall`).

3.1.4 `@multidef`

If the user wants to define a macro with many definitions, he may use `@multidef`. This command has two forms. The most general is:

```
@MULTIDEF <macro handle>
.. Lines of definitions ..
   .         .
   .         .
   .         .
@ENDMULTIDEF
```

If the number of definitions is not large enough to warrent using multiple lines, then the syntax is:

```
@MULTIDEF <macro handle> <def1> <def2> . . . <defN>
```

Note that if there are any non-whitespace characters after the macro handle name, extract will assume the user wants the single-line form (and therefore not look for a matching `@ENDMULTIDEF`).

Definitions may be seperated by whitespace(s) and/or a comma. If the user wishes to have a space included in one of the definitions, he must use the extract's sticky space, `@^` (eg. if you want `Abe Lincoln` to be taken as one definition not two, you would put `Abe@^Lincoln`).

Therefore, these three code segments are equivalent:

1.
```
@define music  @Pearl Jam@
@define music  @Cake@
@define music  @Lyle Lovitt@
```

2. `@multidef music  Pearl@^Jam Cake, Lyle@^Lovitt`

3. ```
   @multidef music
   Pearl@^Jam, Cake
   Lyle@^Lovitt
   @endmultidef
   ```

It is not immediately obvious why a user would want to assign multiple definitions to one handle. We will see that this command is most useful when combined with `@whiledef`, which is discussed in section 3.6.1.

3.1.5 Automatic macros

Each key given as an argument to extract generates an automatic macro. If we say `extract -b bob.base sys=sun4 type=double`, the macros `@(@sys)` and `@(@type)` are automatically defined to be `sun4` and `double`, respectively.

3.1.6 Macro modifiers

Changing case Macros may take two case modifiers. `@up@(handle)` causes the replacement string to be upcased, while prefixing the handle with `@low` will cause it to be lowcased. With neither prefix, the case of the replacement string is left as is. Example:

```
@define bob @This is a String With Mixed Case@
@(bob)
UPCASED  :  @up@(bob)
lowcased :  @low@(bob)
```

Extracted code:

```
This is a String With Mixed Case
UPCASED  :  THIS IS A STRING WITH MIXED CASE
lowcased :  this is a string with mixed case
```

Determining length Any macro prefixed with `@len` resolves to the length of its replacement string, rather than its replacement string. For instance:

```
@define mac1 @012345@
@define mac2 @This is a replacement string@
@(mac1) : @len@(mac1)
@(mac2) : @len@(mac2)
```

When extracted produces:

```
012345 : 6
This is a replacement string : 28
```
```

Formatting Macros can be formatted. The user should specify the number of columns the macro should take up, and whether the macro should be left justified, right justified, or centered within those columns. Let the number of columns specified by N. If the macro length $> N$, then the first N characters of the macro are chosen, and the rest are truncated. The syntax of this formatting is `@N[l,r,c]@(handle)`. If case is modified as well as spacing, the case modifier most be the inner modifier (i.e., `@N[l,r,c]@[up,low]@(handle)`). Therefore, to center the macro bob, within 10 columns, and make it upcase, we would type `@10c@up@(bob)`.
Simple example:

```
NAME            AGE
===========    ====
@11l@(name)    @4r@(age)
```

Extracted with -def name "Petitet, Antoine" -def age "73":

```
NAME            AGE
===========    ====
Petitet, An      73
```

Extracted with -def name "bob" -def age "9":

```
NAME            AGE
===========    ====
bob               9
```

In order to show a more complex example of formatting, we will use the `whiledef` command, explained in section 3.6.1. This example prints out some of the powers of 2 and 10:

```
@multidef col1 16 8 2 1
@multidef col2 1000 100 10 1
POWER of:       2       10
            =====    =====
@whiledef col1
            @5r@(col1)    @5r@(col2)
   @undef col2
@endwhile
```

Extracted code:

```
POWER of:       2       10
            =====    =====
                1        1
                2       10
                8      100
               16     1000
```

3.2 Testing for definitions (@ifdef)

Extract allows the user to test if a certain macro has been defined or not. The structure is:

```
@IFDEF [!] <macro handle>
.. IFDEF CODE ..
@ENDIFDEF
```

Normally, the IFDEF CODE is extracted if the macro handle is defined. If the not operator (!) is applied, however, the IFDEF CODE is extracted if the macro handle has not been defined.
Here are a couple of examples:

```
@IFDEF DEBUG
   @print Everything still working!!
@ENDIFDEF

@IFDEF ! type
   @define type @integer@
@ENDIFDEF
```

3.3 Macro if (@mif, @endmif)

This is used to test the value of a macro substitution. The form is: @mif◇<mac1>◇<comp>◇<mac2>, where <mac1> and <mac2> are either macro handles, or, if they begin with " (no trailing quote should be used), string constants, and comp is one of the options shown in the table below:

comp	MEANING
=	string indicated by mac1 is exactly equal to that of mac2
!	string indicated by mac1 is not equal to that of mac2
~	string indicated by mac1 is a substring to that of mac2

If we have the following basefile:

```
@mif sub1 = "hello
   string is hello
@endmif
@mif "he ~ sub1
   string contains he
@endmif
@mif sub1 = sub2
   sub1 equals sub2
@endmif
@mif sub1 ! sub2
   sub1 not equal to sub2
@endmif
```

And we issue:

```
extract -b tst2.b -def sub1 hello -def sub2 goodbye
   string is hello
   string contains he
   sub1 not equal to sub2

extract -b tst2.b -def sub1 joe -def sub2 joe
   sub1 equals sub2
```

3.4 Integer expressions (`@iexp`)

Performs integral operations, and defines a macro which expands to the result. The syntax is `@iexp`◇`<exp>`, where `<exp>` is a mathematical expression given in reverse Polish notation (i.e., stack based). For example, `@iexp bob 1 4 -`, creates a macro `bob` defined as 3. Therefore, any appearance of `@(bob)` in the code would be replaced by 3. The following integer operations are supported:

+: X and Y are popped from stack, and stack receives $X + Y$.

−: X and Y are popped from stack, and stack receives $X - Y$.

∗: X and Y are popped from stack, and stack receives $X * Y$.

/: X and Y are popped from stack, and stack receives X/Y.

%: X and Y are popped from stack, and stack receives MOD(X, Y).

A: X is popped from stack, and stack receives ABS(X).

|: X and Y are popped from stack, and stack receives $X|Y$.

&: X and Y are popped from stack, and stack receives $X\&Y$.

∧: X and Y are popped from stack, and stack receives $X \wedge Y$.

R: X and Y are popped from stack, and stack receives $X >> Y$.

L: X and Y are popped from stack, and stack receives $X << Y$.

In addition, `@iexp` also can perform comparisons, which result in 1 if true, or 0 if false:

=: X and Y are popped from stack, and stack receives $X == Y$.

!: X and Y are popped from stack, and stack receives $X \neq Y$.

e: X and Y are popped from stack, and stack receives $X \leq Y$.

E: X and Y are popped from stack, and stack receives $X \geq Y$.

Note that expressions result in macros, which are usually defined on a stack. However, `@iexp` macro definitions always result in replacements, and thus do not build up a stack as normal.

If you want to do only one operation at a time (as opposed to utilizing the reverse polish notation to do multiple operations), then the operand order can be summarized as:

OPERATION	RESULT
`@iexp RES Y X OP`	RES = X OP Y

See Section 3.4.1 for arithmetic examples, and §3.5.2 for examples of using the comparisons and bit-level operations to build complicated if conditionals.

3.4.1 Arithmetic examples

Here is a simplistic example:

```
   @iexp num1 1 9 -
   @iexp num2 2 3 *
   @iexp num3 @(num1) @(num2) +
@(num2) + @(num1) = @(num3)
   @iexp num3 @(num1) @(num2) %
@(num2) % @(num1) = @(num3)
   @iexp num3 @(num2) @(num1) /
@(num1) / @(num2) = @(num3)
```

Extracted code:

```
6 + 8 = 14
6 % 8 = 6
8 / 6 = 1
```

Here is an example of using @iexp and macro formatting to print a table of powers of numbers:

```
POWER       @10r@(num1)    @10r@(num2)
=====       ==========    ==========
@define col1 @@(num1)@
@define col2 @@(num2)@
@whiledef pow 5 4 3 2 1
@5r@(pow)       @10r@(col1)    @10r@(col2)
   @iexp col1 @(col1) @(num1) *
   @iexp col2 @(col2) @(num2) *
@endwhile
```

When extracted with `extract -def num1 "2" -def num2 "10"`:

```
POWER                2            10
=====       ==========    ==========
    1                2            10
```

```
    2                4           100
    3                8          1000
    4               16         10000
    5               32        100000
```

When extracted with `extract -def num1 "3" -def num2 "5"`:

```
POWER                3             5
=====       ==========    ==========
    1                3             5
    2                9            25
    3               27           125
    4               81           625
    5              243          3125
```

3.5 Integer if (`@iif`, `@endiif`)

This is used to test the value of integer macros. The form is: `@iif◇<mac1>◇<comp>◇<mac2>`, where `<mac1>` and `<mac2>` are either macro handles, or, if they begin with a number, are assumed to be integer constants, and `comp` is one of the options shown in the table below:

`comp`	MEANING
=	integer indicated by `mac1` is equal to that of `mac2`
!	integer indicated by `mac1` is not equal to that of `mac2`
<	integer indicated by `mac1` is less than that of `mac2`
e	integer indicated by `mac1` is $\leq$ than that of `mac2`
>	integer indicated by `mac1` is greater than that of `mac2`
E	integer indicated by `mac1` is $\geq$ than that of `mac2`

3.5.1 Simple examples

So, if we have the basefile:

```
num = @(num)

@iif num = 10
   num is 10
@endiif
@iif num < 20
   num is less than 20
@endiif
@iif num > 8
   num is greater than 8
@endiif
@iif num ! 10
   num is not 10
@endiif
```

We get:

```
extract -b tstiif.b -def num 10
num = 10

   num is 10
   num is less than 20
   num is greater than 8

extract -b tstiif.b -def num -1
num = -1

   num is less than 20
   num is not 10
```

3.5.2 Complex conditionals using @iexp

Unfortunately, `@iif` takes only a single comparison. So how do you accomplish something like:

```
   if ((x == 5 && y == 6)) || z == -1
```

Since `@iif` can handle only a single comparison, you can instead use `@iexp`'s comparison and bit-level operations to do this. We could write this in the dumbest way possible as:

```
iexp k @(x) 5 =
@iexp j @(y) 6 =
@iexp k @(k) @(j) &
@iexp j @(z) -1 =
@iexp k @(k) @(j) |
@iif k = 1
   TRUE: ((x == 5 && y == 6)) || z == -1
@endiif
```

Or we can express these same operations more concisely using reverse polish notation:

```
@iexp k @(x) 5 = @(y) 6 = & @(z) -1 = |
@iif k = 1
   TRUE: ((x == 5 && y == 6)) || z == -1
@endiif
```

Is it possible to more straightforward and obvious than this? In case it is, Table 3.5.2 shows more examples of setting up complex conditionals, assuming the `@iexp` is followed by an `@iif cn = 1`.

3.6 Looping

3.6.1 Macro definition looping (`@whiledef`, `@endwhile`)

Extract allows for looping upon macros. The loop structure is:

comparison	@iexp equivalent
(i == mu-1 && i >= j)	@iexp cn 1 @(mu) - @(i) = @(j) @(i) E &
(i%2 == 1 || i > 8)	@iexp cn 1 @(i) & 8 @(i) > |
((x == 5 && y == 6)) || z == -1	@iexp cn 5 @(x) = 6 @(y) = & -1 @(z) = |

Table 1: Example @iexp to compute complicated or compound ifs

```
@WHILEDEF <while macro handle> [<def1> . . . <defN>]
.. LOOP BODY ..
@ENDWHILE
```

The definitions are optional. Note there are no surrounding @ @ for the definition here, as there are for `@define`. If spaces are significant, the user must use extract's sticky space, `@^`. In case it is not obvious, there is an implicit `@undef <while macro handle>` at the `@ENDWHILE`.

This is a true while loop, i.e., the condition is tested at the top. A while with no definition is not executed at all. At the moment, `@extract`'s within a while loop will not properly inherit input files, so all `@extract`'s within the loop should have a `-b <basefile>` explicitly given.

The whiledef is handy for many repetitive tasks. It can be used to make a self-extracting basefile. Here is an example which extracts all precisions of a routine to one file:

```
@whiledef type int sreal dreal scplx dcplx
   @extract -b myfile.base -o myfile.f type=@(type) -fmodeA
@endwhile
```

Remember that the definitions are based on a stack, so you must reverse order if it is important. For instance, if you want a loop that counts from 1 to 4, you should write `@whiledef count 4 3 2 1`.
Here is an example of using `@whiledef` to create a C header file for a function which exists in 3 precisions:

```
@multidef type double float int
@whiledef pre d s i
@(type) @(pre)sum(int N, @(type) *X);
   @undef type
@endwhile
```

Extracted code:

```
int isum(int N, int *X);
float ssum(int N, float *X);
double dsum(int N, double *X);
```

3.6.2 Integer value looping (`@iwhile`, `@endiwhile`)

Extract allows for looping over integer values. The general form is:

```
@IWHILE ARG1 COND ARG2
.. LOOP BODY ..
@ENDIWHILE
```

ARG1 and ARG2 may be either integers or macro handles. If they are macro handles, they must not begin with a number. COND is one of =, !,>,<. Here's a simple example:

```
@define i @1@
@iwhile i < 4
   loop count = @(i)
   @iexp i 1 @(i) +
@endiwhile
@undef i
```

When extracted, you get:

```
test. ~/Base/tool/ext3.1 -b tst.b
   loop count = 1
   loop count = 2
   loop count = 3
```

For a more complex example, you might want to write a dot product routine which could be extracted with arbitrary loop unrolling and number of dot product accumulaters. You can do this in extract with:

```
@TYPE SREAL
   @define type @float@
@TYPE DREAL
   @define type @double@
@TYPE !
@ifdef ! nacc
   @define nacc @4@
@endifdef
@ifdef ! nu
   @define nu @4@
@endifdef
@(type) mydot(const int N, const @(type) X, const @(type) Y)
{
   int i;
   const int n = N / @(nu);
@declare "   register @(type) " y n ";"
   @define i @0@
   @iwhile i < nacc
      acc@(i)=0.0
      @iexp i 1 @(i) +
   @endiwhile
   @undef i
```

```
@enddeclare

   for (i=n; i; i--, X += @(nu), Y += @(nu))
   {
   @define i @0@
   @iwhile i < nu
      @iexp acc @(nacc) @(i) %
      acc@(acc) += X[@(i)] * Y[@(i)];
      @iexp i 1 @(i) +
   @endiwhile
   }
   for (i=N-(n*@(nu)); i; i--, X++, Y++) acc0 += *X * *Y;

@define i @1@
@iwhile i < nacc
   acc0 += acc@(i);
      @iexp i 1 @(i) +
@endiwhile

   return(acc0);
}
```

Which, when extracted from `dot.b` could be:

```
test. ext3.1 -b dot.b type=dreal -def nu "8" -def nacc "4"

double mydot(const int N, const double X, const double Y)
{
   int i;
   const int n = N / 8;
   register double acc0=0.0, acc1=0.0, acc2=0.0, acc3=0.0;

   for (i=n; i; i--, X += 8, Y += 8)
   {
      acc0 += X[0] * Y[0];
      acc1 += X[1] * Y[1];
      acc2 += X[2] * Y[2];
      acc3 += X[3] * Y[3];
      acc0 += X[4] * Y[4];
      acc1 += X[5] * Y[5];
      acc2 += X[6] * Y[6];
      acc3 += X[7] * Y[7];
   }
   for (i=N-(n*8); i; i--, X++, Y++) acc0 += *X * *Y;

   acc0 += acc1;
```

```
   acc0 += acc2;
   acc0 += acc3;

   return(acc0);
}
```

Note that since the condition of the `@iwhile` is tested at the top of the loop, it can be used as in `if` statement (i.e., if the condition is initially false, the body of the loop will not appear in the output).

3.7 Skipping lines (`@skip`, `@beginskip`, `@endskip`)

It is sometimes useful to have blocks of text that are *never* included regardless of what commandkeys have been chosen. Consider, for instance, when you have rewritten a large section of code. Until you are confident the new code works correctly, you will want to keep the old code around, but you won't want to extract it. This is done using the `@beginskip` and `@endskip` commands. If extract finds a `@beginskip`, it skips all following lines until an `@endskip` is found. This command can also be useful if one wants to have basefile comments (explaining what the keys mean, etc). Single lines may be skipped using the `@skip` command.

3.8 Indenting (`@beginindent @endindent`)

It is often useful to have certain lines of code indented for some extractors, but not for others. An example is where one extractor gets the code inside of an IF statment, while other extractors do not have the IF at all. The command that accomplishes this is:
`@beginindent`$\diamond\mathcal{S}\diamond\mathcal{N}$
Where $\mathcal{S}$ equals the column to start the indention at (i.e., for fortran you will want to leave the first 6 columns alone, as they have special meaning. You would therefore set $\mathcal{S}$= 7.), and $\mathcal{N}$ indicates the number of spaces to indent by. The indention is stopped by the `@endindent` command. Indents can be nested, and they will be applied in the order of their nesting. I.e., the outer `@beginindent` is applied, and then second outermost, and so on. Negative N's are allowed. Note that this command replaces tabs with spaces before indenting code. Example:

```
@ROUT WantIf
       if (alpha .ne. 1) then
       @beginindent 7 3
@ROUT !
       do i = 1, n
          x(i) = alpha * x(i)
       end do
@ROUT WantIf
       @endindent
       endif
@ROUT !
```

Extracted with `rout=WantIf`:

```
if (alpha .ne. 1) then
   do i = 1, n
      x(i) = alpha * x(i)
   end do
endif
```

Extracted with `rout=DontWantIf`:

```
do i = 1, n
   x(i) = alpha * x(i)
end do
```

3.9 Extending lines (`@\`)

Extract provides a primitive output line extension mechanism. This command cannot be used to extend basefile commands (with the exception of keylines, as discussed in Section 3.16). For instance:

```
@TYPE DREAL
   DI = @\
@TYPE SREAL
   SI = @\
@TYPE !
1.0
```

results in (assuming `type=dreal`): `DI = 1.0`.

3.10 Abnormal extract exit (`@abort`)

Used to halt extraction with error message. The syntax is `@abort◇<error message>`. It exits all extracts: even if it happens several extracts down, all extracts are stopped. `<error message>` is printed out prior to exiting.

3.11 Printing extract messages (`@print`)

This command prints a message to stderr. The syntax is `@print◇<message>`. The main use of this routine is debugging your extract commands.

3.12 Echoing unproccessed lines (`@echo`)

This command sends its argument to the output file with no extract processing other than macro substition. The syntax is `@echo◇<line>`. The main use of this routine is putting extract commands in an output file. An example:

```
@echo @define evil @Bill@
```

Writes to the output file the line `@define evil @Bill@`.

3.13 Using the System (@system)

This command can be used to escape to the system and perform some system call such as might be given on the command line. The syntax is `@system◇"<command>"`. If you are using extract to extract your files, you might want to remove an old copy of a routine, for instance. This could be done by:

```
@system "rm -f @(outdir)/bob.f"
```

3.14 (@declare)

This command can be used in various ways, as discussed in the following subsections. Its syntax is:

```
@DECLARE◇"<start string>"◇[<Continue>◇<Alphabatize>◇[<"end string">◇]<Indent
Column>]
.. variables ..
       ⋮
@ENDDECLARE
```

`<start string>` is placed at the beginning of the line. The next two arguments need not be specified. If `<Continue>` exists and is set to `N`, the start string will be repeated on each line, rather than extending the original line (the default). If `<Alphabatize>` exists and is set to `N`, the variables will not be alphabatized (the default). If `<Indent Column>` exists and is a positive integer, the code will be indented to `<Indent Column>` rather than the length of the `<data type>` string (the default). The user can have a one time string printed after the last parameter by supplying an end string.

For this command to work correctly, it is important that the user specifies the language he is using (C, Fortran77, Fortran90, or Makefile). This tells extract how to extend lines. Also, the command line option `-lnlen` should be used to tell extract how long a line it is permitted to make. If `-lnlen` is not specified on the command line, extract will assume 71 columns for maximal line length for Fortran77, and 80 for all other languages.

If a user has a space in a variable which should not be used to split variables on, the user must use the sticky space `@^` (eg., `int@^i` will be interpreted as one word, and output as `int i`).

3.14.1 Variable declaration

Some programming styles insist that variables be declared in alphabetic order. This can be a pain in extract, when the variable name varies depending on macros or keys. `@declare` can help with this.

For variable declaration, `<start string>` should be the data type being declared, with any spacing the user likes. For instance, in Fortran77, this might be `"      INTEGER "`.

A couple of quick examples may help explain this rather obtuse function:

Assume the language is fortran, and we have the following code with line length set to 40.

```
@declare "      integer "
   ii, kk, k, y, n, buttugly, bob, joe
   idiot, dion, ruth
@enddeclare
```

The ouput from this code would be:

```
      integer bob, buttugly, dion,
     $        idiot, ii, joe, k, kk, n,
     $        ruth, y
```

Now, assume the language is C, and again line length is set to 40.

```
@declare "   int " n y
   ii, kk, k1, k2, k12, y, n, buttugly, bob, Joe
   idiot, dion, ruth
@enddeclare
```

Produces the code:

```
   int Joe, bob, buttugly, dion, idiot;
   int ii, k1, k12, k2, kk, n, ruth, y;
```

3.14.2 Typesetting a subroutine call/declaration

Another use is in typesetting a subroutine call. For example, if the basefile code is:

```
@declare "      call bob( " y n " )"
      Pval, Qval, mAval, nAval,
      mbAval, nbAval, rsrcAval, csrcAval, mBval, nBval,
      mbBval, nbBval, rsrcBval, csrcBval, Mval, Nval,
      IAval, JAval, IBval, JBval
@enddeclare
```

Assuming the language is Fortran77, the code produced is:

```
      call bob( Pval, Qval, mAval, nAval, mbAval, nbAval, rsrcAval,
     $          csrcAval, mBval, nBval, mbBval, nbBval, rsrcBval,
     $          csrcBval, Mval, Nval, IAval, JAval, IBval, JBval )
```

This command is also handy for prototyping in ANSI C. Up until now, I've been showing simple examples; this next one is how I actually use this command myself. The following lines generate the ANSI C prototypes for the C interface to the level 3 BLAS:

```
@whiledef rout gemm symm hemm syrk herk syr2k her2k trmm trsm
   @addkeys rout=@(rout)
   @ROUT HEMM HERK HER2K
      @multidef pre z c
      @multidef type void void
```

```
      @multidef styp void@^* void@^*
   @ROUT ! HEMM HERK HER2K
      @multidef pre z c d s
      @multidef type void void double float
      @multidef styp void@^* void@^* double@^ float@^
   @ROUT !
   @whiledef pre
      @declare "void cblas_@(pre)@(rout)(" y n ");"
         @ROUT SYMM HEMM TRMM TRSM
            enum@^CBLAS_SIDE@^side enum@^CBLAS_UPLO@^uplo
         @ROUT SYRK HERK SYR2K HER2K TRMM TRSM
            enum@^CBLAS_TRANSPOSE@^trans
         @ROUT GEMM
            enum@^CBLAS_TRANSPOSE@^transA
            enum@^CBLAS_TRANSPOSE@^transB
         @ROUT !
         @ROUT TRMM TRSM `enum@^CBLAS_DIAG@^diag`
         @ROUT GEMM SYMM HEMM TRMM TRSM `int@^M`
         int@^N
         @ROUT GEMM SYRK HERK SYR2K HER2K `int@^K`
         @(styp)alpha @(type)@^*A int@^lda
         @ROUT ! SYRK HERK `@(type)@^*B int@^ldb`
         @ROUT ! TRMM TRSM `@(styp)beta @(type)@^*C int@^ldc`
      @undef type
      @undef styp
   @endwhile

   @killkeys rout
@endwhile
```

The extracted (setting the language to C) is :

```
void cblas_strsm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                 enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                 float alpha, float *A, int lda, float *B, int ldb);
void cblas_dtrsm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                 enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                 double alpha, double *A, int lda, double *B, int ldb);
void cblas_ctrsm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                 enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb);
void cblas_ztrsm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                 enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb);

void cblas_strmm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
```

```
                enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                float alpha, float *A, int lda, float *B, int ldb);
void cblas_dtrmm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                double alpha, double *A, int lda, double *B, int ldb);
void cblas_ctrmm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                void *alpha, void *A, int lda, void *B, int ldb);
void cblas_ztrmm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo,
                enum CBLAS_TRANSPOSE trans, enum CBLAS_DIAG diag, int M, int N,
                void *alpha, void *A, int lda, void *B, int ldb);

void cblas_cher2k(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha,
                 void *A, int lda, void *B, int ldb, void *beta, void *C,
                 int ldc);
void cblas_zher2k(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha,
                 void *A, int lda, void *B, int ldb, void *beta, void *C,
                 int ldc);

void cblas_ssyr2k(enum CBLAS_TRANSPOSE trans, int N, int K, float alpha,
                 float *A, int lda, float *B, int ldb, float beta, float *C,
                 int ldc);
void cblas_dsyr2k(enum CBLAS_TRANSPOSE trans, int N, int K, double alpha,
                 double *A, int lda, double *B, int ldb, double beta,
                 double *C, int ldc);
void cblas_csyr2k(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha,
                 void *A, int lda, void *B, int ldb, void *beta, void *C,
                 int ldc);
void cblas_zsyr2k(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha,
                 void *A, int lda, void *B, int ldb, void *beta, void *C,
                 int ldc);

void cblas_cherk(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha, void *A,
                int lda, void *beta, void *C, int ldc);
void cblas_zherk(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha, void *A,
                int lda, void *beta, void *C, int ldc);

void cblas_ssyrk(enum CBLAS_TRANSPOSE trans, int N, int K, float alpha,
                float *A, int lda, float beta, float *C, int ldc);
void cblas_dsyrk(enum CBLAS_TRANSPOSE trans, int N, int K, double alpha,
                double *A, int lda, double beta, double *C, int ldc);
void cblas_csyrk(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha, void *A,
                int lda, void *beta, void *C, int ldc);
void cblas_zsyrk(enum CBLAS_TRANSPOSE trans, int N, int K, void *alpha, void *A,
                int lda, void *beta, void *C, int ldc);
```

```
void cblas_chemm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb, void *beta,
                 void *C, int ldc);
void cblas_zhemm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb, void *beta,
                 void *C, int ldc);

void cblas_ssymm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 float alpha, float *A, int lda, float *B, int ldb, float beta,
                 float *C, int ldc);
void cblas_dsymm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 double alpha, double *A, int lda, double *B, int ldb,
                 double beta, double *C, int ldc);
void cblas_csymm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb, void *beta,
                 void *C, int ldc);
void cblas_zsymm(enum CBLAS_SIDE side, enum CBLAS_UPLO uplo, int M, int N,
                 void *alpha, void *A, int lda, void *B, int ldb, void *beta,
                 void *C, int ldc);

void cblas_sgemm(enum CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB,
                 int M, int N, int K, float alpha, float *A, int lda, float *B,
                 int ldb, float beta, float *C, int ldc);
void cblas_dgemm(enum CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB,
                 int M, int N, int K, double alpha, double *A, int lda,
                 double *B, int ldb, double beta, double *C, int ldc);
void cblas_cgemm(enum CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB,
                 int M, int N, int K, void *alpha, void *A, int lda, void *B,
                 int ldb, void *beta, void *C, int ldc);
void cblas_zgemm(enum CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB,
                 int M, int N, int K, void *alpha, void *A, int lda, void *B,
                 int ldb, void *beta, void *C, int ldc);
```

3.14.3 Makefiles

Finally, `@declare` can be very useful in Makefile maintainence. The following lines:

```
@declare "dobj = " y n
   dgemm dsymm dsyrk dsyr2k dtrmm dtrsm
   dgemv dgbmv dsymv dsbmv dspmv dtrmv dtbmv dtpmv dtrsv dtbsv dtpsv
   dger dsyr dspr dsyr2 dspr2
@enddeclare
```

When extracted with language set to make (`-langM`):

```
dobj = dgemm dsymm dsyrk dsyr2k dtrmm dtrsm dgemv dgbmv dsymv dsbmv dspmv \
       dtrmv dtbmv dtpmv dtrsv dtbsv dtpsv dger dsyr dspr dsyr2 dspr2
```

3.15 Basefile Extract

3.15.1 `@extract`

Basefiles can contain extract commands. An extract inside another extract inherits all options from the calling extract. That means that `@extract`, appearing unadorned in a basefile, causes no change: basefile, output, flags, etc. are all inherited. The most common use of this command is simple include file usage. Here one wants to inherit everything (especially key defs and outfile) from the calling extract, but specify a new input file. This is done by `@extract -b <include file>`.

If any keys are specified, key inheritence is lost. This means that with the command: `@extract -b bob.base type=single` bob.base would have only one key (`@type`) defined, even if this line appeared in a basefile where, for instance, `@sys` was also defined. The user can make it so key inheritence is never lost by using the `-addkeys` flag.

If flag inheritence is not desired, they must be turned off one by one using the 0 argument (e.g. -case0 turns off changing the case).

Input and ouput files are inherited, and they are changed by specifying `-b <basefile>` or `-o <outfile>`, as seen above.

3.15.2 `@endextract`

This command ends the current extract.

3.16 Additional Keyline Commands

3.16.1 Extending Keylines

Keylines can be extended to the next line of text by adding `@\` as the last argument. For example:

```
@TYPE SREAL @\
      DREAL
```

Is exactly equivalent to:

```
@TYPE SREAL DREAL
```

3.16.2 Dynamically adding/removing keys (`@addkeys` & `@killkeys`)

`@addkeys` defines a new extract key. Its form is:
`@addkeys◇<keyhandle1>=<match1> ...◇<keyhandleN>=<matchN>`
Keys, like macros, are on a stack.

`@killkeys` removes the indicated keys from extract use. Its form is:
`@killkeys◇<keyhandle1> ...◇<keyhandleN>`

Note that the keystate of a file cannot be changed by files it extracts. Thus an `@addkeys` or `@killkeys` in an extracted file will not effect the parent file's keys.

Here is a simple example showing how to use `@addkeys` to make a self-extracting file with types:

```
@whiledef pre d s i
   @addkeys pre=@(pre)
      @pre d
         @define type @double@
      @PRE s
         @define type @float@
      @PRE i
         @define type @int@
      @PRE !
   @killkeys pre
@(type) @(pre)sum(int N, @(type) *X)
{
   int i;
   @(type) sum=0.0;
   for (i=0; i != N; i++) sum += X[i];
   return(sum);
}

@endwhile
```

When extracted with no arguments:

```
int isum(int N, int *X)
{
   int i;
   int sum=0.0;
   for (i=0; i != N; i++) sum += X[i];
   return(sum);
}

float ssum(int N, float *X)
{
   int i;
   float sum=0.0;
   for (i=0; i != N; i++) sum += X[i];
   return(sum);
}

double dsum(int N, double *X)
{
   int i;
   double sum=0.0;
   for (i=0; i != N; i++) sum += X[i];
```

```
   return(sum);
}
```

Here is an example of using @addkeys coupled with several other command to print a table of powers of numbers, where the numbers and the max power to go to are all variables (notice that this example also demonstrates how to implement a general `do i = 1, n, inc` loop in extract):

```
POWER       @10r@(num1)    @10r@(num2)
=====       ==========    ==========
@define col1 @@(num1)@
@define col2 @@(num2)@
@define i @1@
@iexp maxpow @(maxpow) 1 +
@addkeys maXPow=@(maxpow)
@whiledef KeepOn TRUE
@5r@(i)       @10r@(col1)    @10r@(col2)
   @iexp col1 @(col1) @(num1) *
   @iexp col2 @(col2) @(num2) *
   @iexp i @(i) 1 +
   @MAXPOW ! @(i)
      @define KeepOn @TRUE@
   @MAXPOW !
@endwhile
@killkeys mAxpow
```

If extracted by `extract -b tst2.b -def num1 "3" -def num2 "5" -def maxpow "4"`:

```
POWER               3            5
=====       ==========    ==========
    1               3            5
    2               9           25
    3              27          125
    4              81          625
```

If extracted by `extract -b tst2.b -def num1 "2" -def num2 "8" -def maxpow "6"`:

```
POWER               2            8
=====       ==========    ==========
    1               2            8
    2               4           64
    3               8          512
    4              16         4096
    5              32        32768
    6              64       262144
```

If you understand the above code, you are entitled to the title “extract ninja”.

3.16.3 Keylines with a one line scope

The case where a user wants to have only one line controlled by a keyline occurs often enough that it is supported seperately from regular keylines. Its syntax is: `@<keyhandle>◇[!]◇key1◇key2◇ ...keyn◇‘<line>‘`

Example:

```
   everybody gets this
@special true ‘      only special people get this‘
   everybody gets this too
```

If extracted with `special=true`:

```
   everybody gets this
      only special people get this
   everybody gets this too
```

If extracted with `special=false`:

```
   everybody gets this
   everybody gets this too
```

3.16.4 Working with the keyarg stack (`@push`, `@pop` & `@peek`)

A regular keyline has the syntax `@<keyhandle>◇[!]◇key1◇key2◇...keyn`, as we have previously seen. Let us define `[!] key1◇key2◇...keyn` to be *keyargs*. Keyargs may be placed on a stack. If the user wishes to save the present keyargs, this is accomplished via the `@push` command. The syntax is: `@<keyhandle>◇@push`.

The previously pushed keyargs may be retrieved via the `@pop` command. Its syntax is: `@<keyhandle>◇@pop`. The keyargs become the active keyargs for extract matching, and are taken off of the keyargs stack.

`@peek` is the same as `@pop`, but the keyargs are not taken off of the stack. It is therefore equivalent to popping followed by a push. Its syntax is: `@<keyhandle>◇@peek`.
Example:

```
@TYPE SREAL DREAL SCPLX DCPLX
@TYPE @push
   All types get this line
@TYPE SREAL DREAL
   Real only gets this
@TYPE @peek
   Back to all types
@TYPE SCPLX
   Only single precision complex
@TYPE @pop
   Back to everybody
```

extracted with `type=sreal`:

```
All types get this line
Real only gets this
Back to all types
Back to everybody
```

extracted with `type=scplx`:

```
All types get this line
Back to all types
Only single precision complex
Back to everybody
```

3.16.5 Accepting or rejecting additional keyargs (+ or -)

The user may wish to begin accepting an additional keyarg(s), without otherwise changing the key state. This is accomplished using the `+` command. Its syntax is:
`@<keyhandle>◇+◇key1◇key2◇...keyn`.

The user may wish to begin rejecting an additional keyarg(s), without otherwise changing the key state. This is accomplished using the `-` command. Its syntax is:
`@<keyhandle>◇-◇key1◇key2◇...keyn`.
Example:

```
@TYPE SREAL DREAL
      only real gets this line
@TYPE + SCPLX
      now sreal, dreal and scplx get this line
@TYPE - sreal
      dreal and scplx get this line
@TYPE !
      everybody gets this line
```

Extracted with `type=scplx`

```
      now sreal, dreal and scplx get this line
      dreal and scplx get this line
      everybody gets this line
```

Extracted with `type=sreal`

```
      only real gets this line
      now sreal, dreal and scplx get this line
      everybody gets this line
```

3.17 Extract procedures `@beginproc`, `@endproc`, `@callproc`

Extract has a facility for creating procedures/subroutines. These procedures are roughly equivalent to extracting from another file: the procedure has access to the calling extract's macros, procedures, indentation settings, etc. It can define macros that survive its call,

etc. Procedures are most useful for including repetitive text or basefile commands that for readability reasons, you prefer not to seperate out into their own file.

An extract procedure is defined as:

```
@BEGINPROC <procnam> [<arg1> ... <argN>]
   PROCBODY
@ENDPROC
```

And called as:

```
@CALLPROC <procnam> [<arg1> ... <argN>]
```

The arguments are optional (i.e. a procedure with no arguments is fine), but the number of arguments must match the number the procedure is called with (extract will issue an error if you call with the wrong number of arguments). Here's an example procedure that defines a caller-selected macro handle to the names of the Level 3 BLAS, and uses that to rename the F77BLAS so that they can be called from C:

```
@beginproc blasdef mnam
   @multidef @(mnam)
      @whiledef pre z c d s
         @(pre)gemm @(pre)symm @(pre)syrk @(pre)syr2k @(pre)trmm @(pre)trsm
      @endwhile
      @whiledef pre z c
         @(pre)hemm @(pre)herk @(pre)her2k
      @endwhile
   @endmultidef
@endproc

#ifdef Add_
   @callproc blasdef l3blas
   @whiledef l3blas
   #define @10l@(l3blas) @(l3blas)_
   @endwhile
#elif defined(UPCASE)
   @callproc blasdef l3blas
   @whiledef l3blas
   #define @10l@(l3blas) @up@(l3blas)
   @endwhile
#endif
```

Extracted, this is:

```
#ifdef Add_
   #define zher2k     zher2k_
   #define zherk      zherk_
   #define zhemm      zhemm_
```

```
   #define cher2k     cher2k_
   #define cherk      cherk_
   #define chemm      chemm_
   #define ztrsm      ztrsm_
   #define ztrmm      ztrmm_
   #define zsyr2k     zsyr2k_
   #define zsyrk      zsyrk_
   #define zsymm      zsymm_
   #define zgemm      zgemm_
   #define ctrsm      ctrsm_
   #define ctrmm      ctrmm_
   #define csyr2k     csyr2k_
   #define csyrk      csyrk_
   #define csymm      csymm_
   #define cgemm      cgemm_
   #define dtrsm      dtrsm_
   #define dtrmm      dtrmm_
   #define dsyr2k     dsyr2k_
   #define dsyrk      dsyrk_
   #define dsymm      dsymm_
   #define dgemm      dgemm_
   #define strsm      strsm_
   #define strmm      strmm_
   #define ssyr2k     ssyr2k_
   #define ssyrk      ssyrk_
   #define ssymm      ssymm_
   #define sgemm      sgemm_
#elif defined(UPCASE)
   #define zher2k     ZHER2K
   #define zherk      ZHERK
   #define zhemm      ZHEMM
   #define cher2k     CHER2K
   #define cherk      CHERK
   #define chemm      CHEMM
   #define ztrsm      ZTRSM
   #define ztrmm      ZTRMM
   #define zsyr2k     ZSYR2K
   #define zsyrk      ZSYRK
   #define zsymm      ZSYMM
   #define zgemm      ZGEMM
   #define ctrsm      CTRSM
   #define ctrmm      CTRMM
   #define csyr2k     CSYR2K
   #define csyrk      CSYRK
   #define csymm      CSYMM
```

```
   #define cgemm      CGEMM
   #define dtrsm      DTRSM
   #define dtrmm      DTRMM
   #define dsyr2k     DSYR2K
   #define dsyrk      DSYRK
   #define dsymm      DSYMM
   #define dgemm      DGEMM
   #define strsm      STRSM
   #define strmm      STRMM
   #define ssyr2k     SSYR2K
   #define ssyrk      SSYRK
   #define ssymm      SSYMM
   #define sgemm      SGEMM
#endif
```