input.F90 — a Fortran90 module for parsing text input

This package includes the following files:

input.f90	The input.F90 module. It is documented by Fortran comments in the file itself, as well as in this document.
test_input.f90	A short program which both illustrates the way in which the module can be used, and also reads a test data file and prints the numbers in it. Note that this program echoes the data lines to the output, which is not the default behaviour.
test_data	A short test file to be read by test_input.f90.
run_test	A shell script to compile and run the test program.
test.output	A copy of the output from the test program. (The test program sends its output to test.out, so it won't over-write this copy.)
doc.pdf	This document.
README	A plain text version of this paragraph.

The main routine in the input module is `read_line`(*eof*), which reads a line of data from the input and parses it into 'words' or items. The logical argument *eof* is set true if end of file is encountered, otherwise false. Items are normally terminated by space or comma, but strings containing these characters may be enclosed in single or double quotes. Any text enclosed in parentheses (round brackets, like this) is ignored unless enclosed in single or double quotes, so that comments may be included in the data file. Several data lines may be concatenated into a single logical line by ending each line but the last with a concatenation sequence — '+++' by default but changeable if required.

Lines beginning with the hash character are treated specially:

# followed by space	Ignore the whole line.
`#concat` *string*	Use the string as the concatenation flag.
`#include` *file*	Read data from the specified file, returning to this file afterwards.
`#revert`	(In an included file) Stop reading this file and return to the data file that included it.

Included files may be nested.

The parsed items are not returned to the calling routine, but are held in module variables and may be read by the following routines. It is possible to read any of the items in the line, but the default is to read them in the order in which they appear. The module variable `item` points to the last variable read, while `nitems` is the number of items on the line.

`call readf`(*a*[,*factor*])
Read the next item in the line as a single or double-precision variable *a*. If the optional argument

factor is present, divide the resulting number by it. (I use this for converting values between external and internal units.) If *factor* is present, it must have the same kind as the variable *a*.

`call readi`(*i*)
Read the next item in the line as an integer *i*.

`call reada`(*string*)
`call readu`(*string*)
`call readl`(*string*)
Read the next item into the character variable *string*. The `readu` routine converts letters in *string* to upper-case, while `readl` converts them to lower-case. The `reada` routine does not carry out any conversion.

If an attempt is made to read beyond the last item on the line, null values are returned and `item` is not incremented further. However there are routines `getf`, `geti` and `geta` which will read an item of the appropriate type, calling `read_line` to read a new logical line if all the items in the current line have already been read.

`call reread`(*k*)
Prepare to read a specified item on the current line. That is, move the item pointer so that the next item read is the one specified:
$k > 0$: Read or reread item k.
$k < 0$: Go back $|k|$ items (but it is not possible to move back past the first item on the current logical line).
$k = 0$: Same as $k = -1$, i.e. reread last item.
The main function of this routine is with argument -1, to reread the last item. This is useful for dealing with optional keywords; an item is read using `readu`, and if it is a recognized keyword the appropriate action is taken. If it is not recognized as a keyword, it is assumed to be a number or string associated with an omitted keyword, and the program reads it again as such, using `reada` in the case of a string to maintain the original case of letters. An example of this usage is given in the test program.

`call input_options`
This routine has a number of arguments, all optional:

`clear_if_null`	If true (default), null values (i.e. blank space between commas or after the last item on the line) are read as zero or the null string. If false, null values are ignored – the existing value is unchanged.
`skip_blank_lines`	If true, blank lines are skipped by the parser. If false (default) they are returned as ordinary data lines that happen to contain no items.
`echo_lines`	If true, each data line is echoed to standard output as it is read. If false (default) this does not happen.
`concat_string`	A concatenation string can be provided to replace the standard one using this argument. Default is "+++".

`error_flag`	An integer value which controls the treatment of errors found in numerical values. If zero (hard errors, default), the program prints an error message and stops. If set to 1 (soft errors) the error message is printed, but the variable being read is set to zero and the program continues. If set to 2, no message is printed, the module variable `nerror` is set to -1, and the program continues. Subsequent errors are hard unless `nerror` is reset to 2.
`default`	If true, reset all options to the default.

`call stream`(*n*)
Take subsequent input from Fortran unit *n*.

The following routines are used internally but are available for use by other routines:

`call upcase`(*string*)
`call locase`(*string*)
Convert letters in *string* to upper or lower case.

`call parse`(*string*)
Parse the string provided in the same way as the `read_line` routine (which uses this routine itself) and leave the details in the buffer as for a line read from the input. Input directives are not interpreted.

`call die`(*string*[,*echo*])
For error messages. Print *string*, print the current input line if the optional logical argument *echo* is present and true, and stop.

`call assert`(*test*,*string*[,*echo*])
Evaluate the logical expression *test*, and stop as for the `die` routine if it is false.

`find_io`(*n*)
This is an integer function, which cycles through I/O units 1–99, starting at unit *n*, until one is found that exists and is not in use, and returns that unit number.